AN OBJECT-ORIENTED APPROACH TO FORMAL SPECIFICATION

by

Graeme Paul Smith
B.E. (Honours)

A thesis submitted to
The Department of Computer Science
University of Queensland
for the degree of

DOCTOR OF PHILOSOPHY

October 1992

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Graeme Smith

Brisbane, October 1992

Abstract

Formal methods for software development are becoming increasingly necessary as software becomes an important part of everyday life. To handle the complexities inherent in large-scale software systems these methods need to be combined with a sound development methodology which supports modularity and reusability. Object orientation, based on the concept that systems are composed of collections of interacting objects whose behaviours are specified by classes, is such a methodology.

This thesis presents the formal specification language Object-Z which is an extension of the formal specification language Z to facilitate specification in an object-oriented style. The major extension in Object-Z is the introduction of the *class schema* which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are objects. Object-Z also supports single and multiple inheritance allowing classes to be reused in the definition of other classes and polymorphism allowing a variable to be assigned to objects of more than one class.

The thesis also presents a set-theoretic model of classes in Object-Z which could form the basis of a full formal semantics. The model, based on the *histories* of a class, i.e. the sequences of states and operations which an object of the class can undergo, facilitates the specification of liveness properties using a temporal logic notation. A fully-abstract model of classes in Object-Z, derived from the history model, is also presented. This model is used to formally define a notion of behavioural compatibility in Object-Z which could form the basis of a theory of class refinement.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr Roger Duke, for his comments and guidance as the ideas in this thesis evolved, and for his unremitting encouragement and help when they weren't evolving so well. I would also like to thank the following people without whose contributions to the development of Object-Z, this thesis would not have been possible – Gordon Rose, Paul King, David Duke and David Carrington. Thanks go also to Ian Hayes, Cecily Bailes and Anthony Lee for their invaluable comments and suggestions.

I would like to acknowledge the generous financial support provided by the Overseas Telecommunications Corporation (OTC) of Australia and the Commonwealth Government of Australia in the form of postgraduate awards. I would also like to acknowledge the Department of Computer Science for support in the form of a research scholarship and tutorship, and for financing attendance at several conferences both in Australia and overseas.

Finally, I would like to acknowledge the support of my family and friends. To my parents, I express my love and gratitude for their encouragement and financial support in my earlier years of study. I also, especially, thank Kim for her continuing love, patience and understanding.

Contents

Chapter 1

Introduction

> *"Only from the alliance of the one, working with and through the other, are great things born."*
>
> — Antoine de Saint-Exupéry
> *The Wisdom of the Sands*, 1948.

Software is being increasingly used in applications which affect our everyday lives. It has found application in the fields of commerce, industry, government, medicine, law and education. In particular, it is being used in a growing number of safety-critical applications ranging from heart pacemakers through to auto-pilots in aircraft. These applications, whose failure could result in injury or even loss of life, demand development techniques which can ensure the production of high-quality, reliable software.

The main approach advocated for making software more reliable is the use of *formal*, or mathematical, methods of software specification, verification and refinement. These techniques, although promising, have not yet been widely adopted. This is due partly to their immaturity and partly to their inability to cope with the complexities inherent in large-scale software systems.

The main approach for handling these complexities in the software itself has been to develop high-level programming languages which support sound modular design and software reusability. A natural culmination of this trend has been the development and growing interest, in the last decade, of programming languages which support the modular design methodology of *object orientation*[80, 20].

Only by adopting similar techniques in languages for the formal specification of software, will it be possible to overcome the problems of scalability. This thesis presents the formal specification language Object-Z[1] which is an extension of the formal specification language

[1]The language Object-Z is, at the time of publication of this thesis, still under development by a team

Z[56, 108, 93] to facilitate specification in an object-oriented style. As a preliminary, this chapter reviews the current state of the art in formal specification and object orientation in Sections 1.1 and 1.2 respectively. Section 1.3 discusses their combination and reviews existing work in this area. Section 1.4 provides an outline of the rest of the thesis.

1.1 Formal Specification

While the need for methods for the formal development of software systems has been recognised for some time[60, 38], it is only within the last few years that they have found increasing application. In 1991, the UK Ministry of Defence prepared a draft standard, 00-55[113], insisting upon the use of formal methods for all military software which may endanger life. A software quality assurance standard, AS 3563[109], is also being developed in Australia.

Formal specification is the first step in the formal development of a software system. It is followed by a series of steps involving verification and refinement which lead to an eventual implementation. The primary role of the formal specification is to provide a precise and unambiguous description of the system as a basis for these subsequent steps.

1.1.1 Uses of formal specifications

A formal specification allows the system designer to verify important properties, resolve ambiguities and detect design errors before system development begins. Without a formal specification, a system would have to be extensively tested after implementation. This alternative is not only expensive, since on failing the tests the system may need to be reimplemented, but also can never guarantee reliable behaviour.

A formal specification also provides a means of communication between the system designer and other persons involved with the system. It acts as a contract between the system designer and the client who wants the system built. It provides a plan (or blueprint) for the system implementor, or programmer, and can form the basis for the user documentation.

It can also provide a means of communication between designers of separate systems which need to be in some way compatible. For example, international standards of communication protocols are formally specified so that telecommunications companies in different countries can build compatible exchanges.

of researchers. This thesis presents a particular version of Object-Z which has been developed by the thesis author and which does not correspond entirely to any previously published version of the language. Other versions of Object-Z can be found in [27, 41].

In the long-term, a formal specification may be used as a reference manual for maintenance and modification of the system. Proposed extensions to the system can be formally specified and their interactions with the existing system determined.

1.1.2 Properties of specification languages

To enable verification of system properties and refinement towards an implementation, a language for formal specification must be mathematically based. Usually this basis is expressed algebraically or in set theory and logic. A formal specification language must also have a well-defined syntax and semantics.

Formal specification languages are generally concerned with specifying *safety* properties. That is, properties that state that something 'bad' doesn't happen[70]. Formal specification languages may, however, also be used to specify *liveness* properties. That is, properties that state that something 'good' does happen[70]. Liveness properties may state that an event is guaranteed to take place, that events occur fairly or that the system is guaranteed to terminate successfully, i.e. it won't run forever. A formal definition of safety and liveness properties is given by Alpern and Schneider[6].

A formal specification language need not, in general, be executable. Some specification languages such as OBJ[51] are executable. However, to manage this, these languages sacrifice other desirable properties such as abstraction and nondeterminism. This issue is discussed in detail by Hayes and Jones[58].

Abstraction allows the specification of a system's functionality independent of any particular implementation. That is, the specification describes what the system does without regard for how it does it. Such specifications are necessarily simpler not having to describe any algorithmic details and are therefore easier to reason about and verify. Abstraction also allows the specification of system interfaces which aid in composing systems.

Although abstraction from implementation details is desirable, it is also desirable to be able to capture these details in a specification language. This allows refinement towards a suitable implementation. A specification language should therefore ideally be capable of specification at many levels of abstraction.

Nondeterminism allows the specification of systems which exhibit more than one response to a given input. Support for nondeterminism is desirable in a specification language since a purely deterministic specification may unnecessarily constrain the choice of possible system implementations.

Another property which is sometimes considered important for a specification language is the ability to capture concurrency. Concurrency is essentially an implementation issue, a concurrent system being one where more than one process is being run simultaneously. To allow refinement towards such an implementation, however, requires the specification language to model the simultaneous occurrence of more than one event. Also, for many

systems, even functionality at the highest level is best captured in terms of a collection of concurrent components. For example, a multiprocessor system is best specified as a collection of concurrently operating processors.

Specification languages should also be usable. That is, despite their formality and mathematical basis, they should be readable and easily comprehended. One way to achieve this is by structuring the specification into independent parts that can then be read and understood in isolation. Methods such as genericity which enable reuse of existing parts of the specification can also aid readability . Readability can also be achieved by including within the specification informal explanatory text and diagrams. These, however, should only assist the understanding of the formal part, not replace it.

1.1.3 Classes of specification languages

A wide range of formal specification languages have been proposed. Most of these languages can be classified as either *property-oriented* or *model-oriented*. Property-oriented languages describe a system implicitly by stating its properties whereas model-oriented languages construct an explicit model of the system.

Property-oriented languages

Property-oriented languages include algebraic languages such as Clear[23] and OBJ, axiomatic languages[71] and temporal logics[92, 84, 72]. The latter languages are particularly suited to specifying liveness properties. As an example of specification with a property-oriented language, consider the following Clear specification of a simple buffering system that can store up to two items.

const *Item* =
  theory
    sorts *item*
    opns $error : item$
  endth

const *Buffer* =
  enrich *Item* **by**
    data sorts *buf*
      opns $empty : buf$
        $in : item, buf \rightarrow buf$
        $out : buf \rightarrow (item, buf)$
      eqns $in(z, in(y, in(x, empty))) = in(y, in(x, empty))$
        $out(in(x, empty)) = (x, empty)$
        $out(in(y, in(x, empty)) = (x, in(y, empty))$
        $out(empty) = (error, empty)$
  enden

The specification starts with a *theory* describing the basic type, or sort, *item*. The operation *error* can be thought of as a function which takes no arguments. It defines a constant of sort *item*. Since all operations in algebraic specification languages are required to be total, such constants are needed as the return arguments of operations applied outside their intended domain.

The theory *Buffer* enriches the theory *Item* with a new sort *buf*, and operations *empty*, *in* and *out*. The keyword **data** indicates that the sort *buf* is defined completely by this theory. That is, all values of *buf* can be generated by applying the theory's operations. The sort *item* is not constrained in this way.

The equations state that the buffer will only hold up to two items and that they are removed on a first-in/first-out basis. If the operation *out* is applied when the buffer is empty, the error constant is returned.

Model-oriented languages

By explicitly modelling systems, model-oriented languages tend to give specifications which are easier to understand than those produced by property-oriented techniques. However, property-oriented techniques, being more abstract, tend to be better suited to reasoning about specified systems.

Model-oriented languages can be further classified into *state-based* and *event-based* languages. State-based languages include an explicit state as part of the model. Transitions are then defined on the state. Event-based languages describe transitions, or events, without reference to an explicit state.

State-based languages include logic-based languages such as Z and VDM[65], as well as languages based on nets[89] and finite state machines[12, 63]. As an example, consider once again a simple two-place buffer. A specification in Z can be given as follows.

$[Item]$

Buffer

$$buf : \operatorname{seq} Item$$
$$\#buf \leqslant 2$$

Init

$$Buffer$$
$$buf = \langle\,\rangle$$

In

$$\Delta Buffer$$
$$in? : Item$$
$$\#buf < 2$$
$$buf' = buf \frown \langle in? \rangle$$

Out

$$\Delta Buffer$$
$$out! : Item$$
$$buf \neq \langle\,\rangle$$
$$buf = \langle out! \rangle \frown buf'$$

The specification consists of the declaration of a basic type *Item*, one state schema *Buffer*, an initial state schema *Init* and two operation schemas *In* and *Out*. Each schema consists of a declaration of some variables together with a predicate relating these variables.

The state schema *Buffer* describes the state of the system as a sequence of items of length two or less. The initial state schema *Init* includes the variable and predicate of *Buffer* and adds a new predicate to indicate that the buffer is initially empty.

The operations schemas *In* and *Out* describe state transitions. The values of variables before a transition are unprimed whereas the values after are primed. The primed and unprimed variables of *Buffer* are introduced by the declaration $\Delta Buffer$.

In has an additional variable *in*? corresponding to an item to be input to the buffer. By convention, the names of input variables end with a question mark. *In* also has a precondition that the length of the buffer must be less than two. The specification says nothing about what happens if the operation is applied when its precondition does not hold. That is, under these conditions the outcome of the operation is unspecified. A more robust specification could be given by incorporating schemas for error handling. Examples of this can be found in Hayes[56].

Out also has an additional variable *out*! this time corresponding to an item to be output from the buffer. By convention, the names of output variables end with an exclamation mark. *Out* has a precondition that the buffer is not empty.

Event-based languages include process algebras such as CSP[61], CCS[81] and ACP[13], as well as languages based on traces[62, 78] and grammars[10]. Such languages are ideally suited to capturing concurrency.

For example, a two-place buffer can be specified in CSP as the concurrent composition of two one-place buffers.

$$B1 = in?x \rightarrow transfer!x \rightarrow B1$$
$$B2 = transfer?y \rightarrow out!y \rightarrow B2$$
$$BUFFER = (B1 \parallel B2) \setminus \{transfer\}$$

The processes $B1$ and $B2$ define traces of events corresponding to the behaviours of the one-place buffers. Each event consists of a channel and a message which is passed on that channel. For example, $in?x$ is the event that passes the message x on input channel *in*. Similarly, $transfer!x$ is the event that passes the message x on output channel *transfer*.

The concurrency operator $\parallel$ is used to construct a process which is composed of the component processes $B1$ and $B2$. Any trace of this process is an interleaving of traces of $B1$ and $B2$ subject to the restriction that synchronisation occurs on the common-named channel *transfer*. Events involving this channel are subsequently hidden in the traces of the process *BUFFER*. When specifying systems in terms of concurrent components it is often necessary to specify channels such as *transfer* which are internal to the system. Hiding the events involving these channels is necessary to prevent them being further constrained by the system's environment.

Heterogeneous languages

Some specification languages do not belong to just one of the above classes. For example, the specification language LOTOS[19, 64] has two distinct parts: a process algebra based on CCS and an abstract data type language based on the algebraic specification language ACT ONE[46].

The Object-Z specification language developed in this thesis also combines two techniques. Being an extension to Z, it is primarily a state-based language but it also has a temporal logic component used to capture liveness properties.

1.2 Object Orientation

The most common approach to handling complexity in large-scale software systems is to decompose them into a number of easily comprehensible parts, or modules. This approach, advocated by Parnas in 1972[88], has motivated the development of several modular programming languages including Ada[11], CLU[75] and Modula-2[121].

Object orientation is a modular design methodology based on constructing systems as collections of interacting components called objects. It originated with the programming language Simula 67[14], an extension of Algol 60 which was concerned with modelling objects in the real world for the purpose of simulation. As a result, object orientation offers a particularly intuitive way of modelling systems.

An object has a state and a set of operations which may act on its state. The state consists of a collection of state variables, or attributes, some of which may designate other objects. In this way, objects may themselves be composed of objects. Objects differ from abstract data types by the presence of their state. This affects their method of data abstraction. Abstract data types abstract away from data by creating a type whose representation is unknown to the user, or client. Objects, on the other hand, abstract away from data by means of a procedural interface. This difference affects extensibility, efficiency, typing and verification as shown by Cook[29].

Object orientation gained rapid popularity as a programming paradigm in the last decade following the advent of Smalltalk-80[53]. Several object-oriented languages have since been developed including Eiffel[80], POOL[9] and FOOPS[50]. Also, several object-oriented extensions to existing programming languages have been developed. The most notable among these are the C extensions, C++[112] and Objective-C[32], and the Lisp extensions, LOOPS[18] and Flavors[24].

1.2.1 Benefits of object orientation

Most of the benefits of the object-oriented approach extend directly from its modularity. The most obvious of these is that it aids understanding. Each of the objects which make up a system can be understood and reasoned about in isolation. Understanding the system in its entirety is then reduced to simply understanding the interactions between its objects.

Object orientation also encourages software reuse since smaller components are more likely to be reused in a system than larger ones. By creating libraries of object templates, or *classes*, software reuse between systems is also possible.

Object orientation also makes it easier to modify or extend software systems. Often simple modifications will only affect one object and these modifications can be made to that object in isolation.

1.2.2 Features of object orientation

There has been much debate as to what "object-oriented" actually means. This is evident from the varying features supported by different object-oriented languages. Most, however, support the notion of classes as a means of encapsulating the state information and operations of objects. Many also support the notions of *inheritance* and *polymorphism*. Inheritance is a mechanism for incremental modification of classes. Polymorphism is the ability of variables to be assigned to objects of more than one class.

Classes

Objects can be grouped into classes which encapsulate their state information and associated operations. Objects with the same class have the same state variables and operations and hence the same behaviour. For example, consider the following Eiffel class *PERSON*[2].

[2]The operation *Create* which is required for initialisation of objects upon creation has been left out of this and subsequent Eiffel classes in this section for reasons of simplicity.

```
class PERSON export
    name, age, birthday
feature
    name : STRING;
    age : INTEGER;
    birthday is
            -- Increment age
        do
            age := age + 1
        end;  -- birthday
end -- class PERSON
```

This class has two attributes *name* of type *STRING* and *age* of type *INTEGER*, and an operation *birthday* which increments *age*. The class itself is not an object but is used as a template to create objects. In this way, a number of distinct objects can be defined which share the code of their class.

A class can be thought of as defining a type such that objects of the class are instances of the type. While some object-oriented languages, including Eiffel, identify a class with the type it defines, it is important to distinguish between these concepts. An object's class describes its (implementational) structure whereas an object's type describes its properties or behaviour. It is possible for objects with identical properties to be structured differently. For example, a stack may be implemented as either an array or a linked list. A discussion of the distinction between classes and types is given in [52].

A class provides a well-defined interface limiting a client's view of an object. Often all, or part, of an object's state is hidden from the client who only sees its interaction with the environment. This enables changes to be made to the state of a class, perhaps to improve efficiency, without affecting the operation of the rest of the system.

Hiding the state of an object also prevents clients from changing attributes in ways other than those defined by the object's operations. This is vital for the correct operation of most objects.

Sometimes, however, it is necessary for the values of certain attributes to be available to clients. In languages such as Smalltalk-80, where the entire state is hidden, separate operations have to be defined for each attribute that may be accessed. Other languages provide mechanisms so that the hiding of attributes (and operations) is left to the implementor of the class. For example, the *export* list of Eiffel and the use of *public*, *protected* and *private* in C++.

The notion of hiding the state of an object is central to the Object-Z language presented in this thesis. An object in Object-Z may only be accessed through the procedural interface defined by its class. The syntax and semantics of classes and object instantiation in Object-Z are presented in Chapter 2.

Inheritance

Inheritance is defined by Wegner and Zdonik[117] as a mechanism for "incremental modification" of classes. Although it was originally introduced into Simula 67 as an organisational tool for classification, its primary use in modern object-oriented programming is as a means of sharing code between classes.

Inheritance allows a class, called a *subclass*, to be constructed from another class, called its *superclass*. A subclass usually represents an extension or specialisation of its superclass. For example, consider the Eiffel class *STUDENT* which inherits *PERSON*.

```
class STUDENT export
    name, age, nb_subjects, birthday, enrol, complete
inherit
    PERSON
feature
    nb_subjects : INTEGER;
    enrol is
            -- Enrol in a new subject
        do
            if nb_subjects < 8 then
                nb_subjects := nb_subjects + 1
            end
        end; -- enrol
    complete is
            -- Complete a subject
        do
            if nb_subjects > 0 then
                nb_subjects := nb_subjects - 1
            end
        end; -- complete
end -- class STUDENT
```

The class *STUDENT* includes, through inheritance, the attributes and operation *birthday* of *PERSON* as well as the additional attribute *nb_subjects*, corresponding to the number of subjects currently studied, and the additional operations *enrol*, corresponding to enrolling in a new subject, and *complete*, corresponding to completing a subject. The operation *enrol* is defined so that a student can only enrol in up to 8 subjects at any time.

As well as addition of attributes and operations, inheritance may also allow existing attributes and operations to be renamed and existing operations to be redefined. For example, the class *HONOURS_STUDENT* inherits student and redefines the operation *enrol* so that up to 10 subjects may be studied at any time.

```
class HONOURS_STUDENT export
    name, age, nb_subjects, birthday, enrol, complete
inherit
    STUDENT redefine enrol
feature
    enrol is
            -- Enrol in a new subject
        do
            if nb_subjects < 10 then
                nb_subjects := nb_subjects + 1
            end
        end; -- enrol
end -- class HONOURS_STUDENT
```

The new definition of *enrol* overrides the existing one in *STUDENT*. Redefinition allows the possibility of *deferred* operations which are only fully defined in the subclasses of the class in which they originally occur. Classes with deferred operations, called deferred classes, cannot be used for object declaration but provide an abstract representation of their subclasses.

Inheritance may also allow attributes and operations to be cancelled. This causes the subclass to be a generalisation, rather than a specialisation, of its superclass. Cancellation of attributes and operations, however, is not supported in most object-oriented languages.

Many object-oriented languages allow a class to inherit from more than one class. This is called multiple inheritance. Notions of single and multiple inheritance in Object-Z are presented in Chapter 3 of this thesis. These notions allow inherited attributes and operations to be renamed and inherited operations to be arbitrarily redefined.

Polymorphism

Polymorphism is described by Meyer[80] as "the ability to take several forms". A number of different kinds of polymorphism that exist in programming languages are discussed by Cardelli and Wegner[26].

In the context of object orientation, polymorphism refers to the ability of a variable to be assigned to objects of more than one class. The classes of objects to which a variable may be assigned are in some way compatible with its declared class.

In many object-oriented languages, polymorphism is restricted to the inheritance hierarchy. That is, an object-valued variable may be assigned to an object of its declared class or any class derived from its declared class by inheritance. For example, a variable declared to be of class *PERSON* could be assigned to an object of any of the classes *PERSON*, *STUDENT* or *HONOURS_STUDENT*.

Associating inheritance and polymorphism requires restricting the modifications allowed by inheritance so that subclasses are in some way compatible with their superclasses. In some languages, such as Smalltalk-80, which don't provide relevant type-checking facilities, it is the responsibility of the program designer to ensure compatibility is maintained through inheritance.

In other languages, such as C++, inheritance is restricted to ensure a type of compatibility known as *signature compatibility*. Informally, a subclass is said to be signature compatible with its superclass if it has all the attributes and operations of the superclass (and perhaps some more). This allows the subclass to be applied in any environment in which its superclass can be applied, i.e. all operation invocation sequences allowed for the superclass are also allowed for the subclass. A formal definition of signature compatibility in Object-Z is presented in Chapter 3 of this thesis.

In Eiffel, inheritance is restricted even further to ensure *behavioural compatibility*. Objects of the subclass can not only be applied in any environment that their superclass can be applied in, but will also respond in a way that their superclass would have responded. This can only be ensured by limiting the redefinition of operations. Eiffel adopts a rule for the redefinition of the pre-conditions and post-conditions of operations based on the notion of *contravariance*. Basically, this requires that the redefined operation be applicable in (possibly) more states than the original operation and, for a given pre-state, result in (possibly) less allowable post-states.

By ensuring behavioural compatibility through inheritance, subclasses in Eiffel are also subtypes. However, just as it is important to distinguish between classes and types, it is also important to distinguish between subclassing and subtyping. Subclassing is concerned with sharing of the *internal* structure of classes whereas subtyping is concerned with specialising the *external* behaviour, or functionality, of classes. The distinction between subclassing and subtyping has been discussed by America[7], Cook *et al.*[30] and Snyder[105].

Behavioural compatibility, however, is an important concept as it determines whether one object can be substituted for another without affecting the behaviour of the rest of the system. This has many practical applications, such as updating the implementation of an object for reasons of efficiency or to add extra functionality, and is also related to the notion of class refinement[114, 119]. A formal definition of behavioural compatibility in Object-Z, which could form the basis of a theory of class refinement, is presented in Chapter 6 of this thesis.

Other features

Other features supported by object-oriented languages include the notions of *delegation* and *object identity*. Delegation is described by Wegner[116] as a "class-independent notion of inheritance". It allows objects to delegate responsibility for performing operations or returning values to other objects. Although delegation can model class-based inheritance

and is therefore more flexible, it is a difficult concept to handle semantically[5]. Delegation will not be considered further in this thesis.

Objects are often regarded as having an identity independent of their state and operations. That is, they are not characterised completely by their class. This enables objects from the same class with identical attributes values to be distinguished. Unlike in many object-oriented languages where object identity is implicit, it must be explicitly modelled in Object-Z. This issue is examined in Chapter 2 of this thesis.

1.2.3 Models of object orientation

To help clarify the fundamental concepts of object orientation, many formal and semi-formal models of object orientation have been developed. Some of these models are in the form of semantics for object-oriented programming languages while others take a more general view of object orientation.

Semantics of object-oriented programming languages

Wolczko[122] presents a denotational semantics of Smalltalk-80 based on the "Virtual Machine" model described by Goldberg and Robson[53]. This model isn't very abstract and includes many implementational details specific to Smalltalk-80 such as the "method lookup" mechanism required for inheritance. This mechanism works by first searching an object's class for a particular method, or operation, and, if it is not found, searching its superclass. This search continues up the class hierarchy until the method is found.

Operational definitions such as this, however, do not always lead to an intuitive understanding of the concepts they are describing. Cook and Palsberg[31] describe a more abstract denotational model of inheritance using *fixed points*. Fixed points allow a recursive function, i.e. one defined in terms of itself, to be transformed into a non-recursive function called a *generator*. In their semantics, Cook and Palsberg represent a class as a recursive function and use the generator of this function to create objects. Inheritance is modelled as an operation on generators. Correctness of the model is proven by showing it is equivalent to an operational semantics based on the "method lookup" mechanism.

Similar models of inheritance are given in the denotational semantics of Smalltalk-80 presented by Kamin[68] and Reddy[94]. Kamin's model represents an object as having a local environment and a reference to a class. Reddy's model is more abstract as it hides the local environment. Objects in this model are represented as *closures*, i.e. data structures containing functions with access to some local state.

Both Kamin and Reddy are motivated by the idea that the denotations of objects should only be detailed enough to explain their externally observable behaviour. A semantics with this property is described as being *fully-abstract*[90, 111, 79]. Yelland[123] shows that

Reddy's model is not fully-abstract as systems which are behaviourally equivalent can be given different semantic denotations. He proposes a fully-abstract algebraic semantics as well as a more "natural" fully-abstract semantics based on state-transition graphs.

A fully-abstract model of classes in Object-Z is presented in Chapter 5 of this thesis. From a theoretical point of view, this model is interesting as it captures the precise meaning of a class independent of its syntactic representation. From a practical point of view, the model allows simpler definitions of behavioural compatibility, and hence subtyping and refinement, to be developed.

General models of object orientation

Much of the language-independent formal work on object orientation has involved the development of relevant type theories[25, 26, 3]. A review of the work in this area is presented by Danforth and Tomlinson[37]. Several general models of object orientation have, however, also been developed.

Wand[115] proposes a model of object orientation which reflects the idea of objects as providing an intuitive or natural way to describe systems. The model is based on the ontological concepts defined in Bunge's "Treatise on Basic Philosophy". The model abstracts away from the notion of operations. Instead, objects are described in terms of their observable attributes and *laws* which constrain the allowed combinations of values of the attributes.

Other, more formal, models of object orientation have been proposed by Cusack[33], Goguen[49] and Ehrich and Sernadas[45]. Cusack presents a language-independent framework for inheritance based on a generic notion of refinement. Classes are represented as sets of objects and inheritance by the subset relation. The definition of classes, therefore, defines the type of inheritance allowed. This model insists that if one class is a subtype of another then it is necessarily also a subclass. In [36], the model is modified to distinguish between inheritance and subtyping.

Goguen models objects in terms of the observations that can be made of them over space and time. This model captures the behaviour of objects, rather than their structure, and also enables the behaviour of a system of objects to be derived from the behaviours of its components. The model is particularly oriented towards capturing concurrency in object-oriented systems.

Ehrich and Sernadas also adopt an approach which describes objects in terms of their behaviour. An object is modelled as a mapping from the trace of operations it undergoes to the corresponding trace of its attribute's values. This approach is combined with the work of Goguen in [44]. Various forms of inheritance and refinement are defined in terms of the combined model.

A behavioural model of classes is presented in Chapter 2 of this thesis as the basis of a full formal semantics of Object-Z. The model is based on representing an object by

the sequence of states it has passed through together with the sequence of events it has undergone. This model facilitates the specification of liveness properties in Object-Z using a temporal logic notation presented in Chapter 4. The model is also used in the derivation of the fully-abstract model of classes presented in Chapter 5.

1.3 Object-Oriented Formal Specification

The benefits derived from object orientation apply not only to software implementation, but to any stage of software development. In particular, object orientation can be used in the formal specification of software as a means of handling the complexity of large-scale systems.

The enhanced structuring and reusability provided by adopting an object-oriented approach not only improves the clarity of specifications but also aids in the subsequent steps of verification and refinement. Properties of objects can be verified locally and then these properties can be used to verify global properties of the system. By choosing to implement in an object-oriented programming language, the specification can be refined to represent the exact structure, in terms of attributes and operations, of each object in the implementation.

Specifying systems as collections of objects, however, generally results in a lower level of abstraction as the consequent structuring is often suggestive of a possible implementation. High-level specifications, however, can be accommodated in such cases by modelling the system as a single object. The class of such an object provides a specification of the system's interface with its environment.

Object-oriented concepts have been incorporated in several specification languages including algebraic specification languages[100, 28], SDL[85], Estelle[101], VDM[102, 120] and CSP[33]. Most work in the area of object-oriented formal specification, however, has involved the specification languages LOTOS and Z.

1.3.1 Object-oriented LOTOS

In general, process algebras offer an intuitive model of the behaviour of objects and are, therefore, ideal for capturing notions of subtyping and behavioural compatibility. Mayr[77] suggests using LOTOS to specify object-oriented systems. The behaviour of an object is specified as a LOTOS process and a system of objects as a parallel composition of such processes. Subtyping is identified as being equivalent to the *extension* relation defined for LOTOS by Brinksma *et al.* in [21].

Similarly, Cusack *et al.*[36] build on the work in [33] to develop an object-oriented interpre-

tation of Basic LOTOS[3]. A class template is specified as a LOTOS process definition and object instantiation is identified as being equivalent to the *conformance* relation defined in [21]. That is, any process which conforms to a class template represents an object of that class. Subtyping is once again identified as being equivalent to the extension relation of Brinksma *et al.*

While subtyping is captured easily in these approaches, a general notion of inheritance is not. Black[15] discusses a notion of inheritance which involves the reuse of one or more LOTOS process definitions within another. However, as Black goes on to show, problems arise with this approach when an inherited process is defined recursively. A syntactic extension to overcome these problem has been subsequently proposed by Rudkin[96].

A more fundamental problem with modelling inheritance in LOTOS, however, is to do with the nature of inheritance itself. As mentioned in Section 1.2.2, inheritance is concerned with the structure of classes and not with the behaviour of their objects. This structure is not captured by a LOTOS process definition.

State-based techniques, on the other hand, are ideal for modelling the structure of a class. Consequently, object-oriented adaptations of these techniques have met with greater success. By adopting a semantics of classes based on the the behaviours of their objects, such techniques can also be used for discussing notions of behavioural compatibility. This is precisely the approach adopted for the semantics of Object-Z presented in this thesis.

1.3.2 Object-oriented Z

The earliest published work on object-oriented Z is that of Schuman and Pitt[98]. They present a variant of Z in which classes are specified as a single state schema and an associated set of operation schemas. The schemas belonging to a particular class are related by their headers. The header of the state schema identifies the class name and the header of each operation schema is prefixed by this name. The primed and unprimed variables of the class state are included implicitly within each class operation.

While some changes are made to the syntax of Z to accommodate this style, the major departure from standard Z is semantic. A class is represented semantically by a set of operation histories. A rule of "historical inference" is introduced which means only the minimum change to a state need be specified in an operation. A state variable not mentioned in an operation's post-condition is not assigned a nondeterministic value, as in standard Z, but simply remains unchanged. Rules of inference for reasoning about class specifications are given in terms of this semantics.

Schuman and Pitt allow inheritance to be modelled by inheriting each schema from another class individually into the corresponding schemas of the new class. Their work is extended in [99] to also include object instantiation and concurrent composition.

[3]Basic LOTOS is standard LOTOS without the abstract data type language ACT ONE.

Hall[55] presents conventions for specifying systems in an object-oriented style in standard Z. These conventions include the use of object identities, a means of expressing the state of a system in terms of its component objects and a means of specifying the effect on a system of an operation on one of its component objects. Extensions to Z to more readily handle these conventions are also suggested.

Cusack and Lai[35] present an extension to Z incorporating the notion of a *template* schema which encapsulates a state schema together with an initial state schema and zero or more operation schemas. A template schema represents a class type, i.e. instances of a template schema are objects of an associated class. Based on the work in [33], various pre-orders are defined on template schemas in an attempt to define a subtyping relation.

This work is extended by Cusack in [34] to consider inheritance in object-oriented Z. Two types of inheritance are defined: *incremental inheritance* for reuse of existing class specifications and *subtyping inheritance* for specifying classes which may be substituted for their inherited classes.

Whysall and McDermid[118] present an approach to object-oriented Z in which each object has two separate specifications: an algebraic *export* specification and a *body* specification in standard Z. The approach is motivated by the use of object specifications during refinement. The export specification provides a description of the object's behaviour without reference to its state and is used to prove properties about the object which may be needed in reasoning about systems in which it is used. The body specification provides a basic description of the object in terms of its state and operations and is used as a basis for its subsequent refinement. Refinement, substitution and composition relations in this framework are presented in [119].

Lano[73] presents an object-oriented extension to Z called Z++. The syntax of Z is extended to allow class types to be defined. A class type has local type and variable declarations and a set of operations. The operations are defined as functions from input parameters to output parameters and adopt the convention that variables not mentioned in their definition are unchanged. A class type may also have generic parameters and may inherit other classes.

Instances of class types are used in standard Z specifications. This allows the modification of data structures without requiring changes to the specification body. The class concept in Z++, therefore, primarily provides a means of separating the implementational details of data structures from the high-level specification of system functionality.

Alencar and Goguen[4] present an object-oriented specification language called OOZE (Object Oriented Z Environment) which combines an extended Z syntax with the algebraic semantics of the object-oriented programming language FOOPS. OOZE supports not only classes but also *modules* which are used to group related classes, *theories* which are used to specify module interfaces and *views* which assert relationships of refinement between modules. Object instantiation as well as inheritance, at both the class and module level, are supported. Specifications in OOZE may be restricted to a subset of the

language so that they are either interpretable, for rapid prototyping, or compilable, for implementation.

Object-Z has been developed independently of these other approaches and has, in fact, influenced some of them. In particular, the work of Cusack has been directly inspired by Object-Z as has the syntax of OOZE. Object-Z supports both object instantiation and multiple inheritance and is currently the only object-oriented adaptation of Z which allows the specification of liveness properties. Object-Z is compared with the other object-oriented adaptations of Z in Chapter 7.

1.4 Thesis Outline

The objective of this thesis is to present the formal specification language Object-Z and to provide a basis for the development of a full formal semantics and theory of refinement for the language. This basis is provided in terms of a set-theoretic model of classes and an associated theory of the behavioural compatibility of objects.

Chapter 2 introduces the notion of a class in Object-Z which groups a single state schema with the operation schemas which can affect its variables. The notation presented was originally developed by the thesis author in collaboration with the authors of [42] and subsequently revised by the thesis author in collaboration with the authors of [27] and [41]. The author's contribution to all aspects of each of these papers was substantial.

A set-theoretic model of classes, based on the sequences of states and operations an object of a class can undergo, is also presented in Chapter 2. This model, initially inspired by the work of Duke and Duke[39], is used to explain the meaning of object instantiation and the initialisation and application of operations to objects of a class.

Chapter 3 presents inheritance and polymorphism in Object-Z. The fundamental ideas presented were developed by the thesis author in collaboration with the authors of [27]. Inheritance in Object-Z is not restricted to maintain compatibility between a class and its subclasses and, hence, it is the responsibility of the specifier to ensure compatibility exists when an inheritance hierarchy is used polymorphically. Rules for maintaining signature compatibility through inheritance are presented in this chapter.

Chapters 4, 5 and 6 present original research carried out solely by the thesis author. Chapter 4 examines the specification of liveness properties in Object-Z. A formal definition of the concepts of safety and liveness are presented as well as the full formal syntax and semantics of a temporal logic notation for specifying liveness properties concerned with the occurrence of both states and events. This notation is incorporated into the syntax and semantics of Object-Z classes.

Chapter 5 presents a fully-abstract model of classes in Object-Z. Within the model, the denotation of a class contains the minimum amount of information such that the denotation of the class of any object which is composed of a number of objects can be determined

from the denotations of the classes of these objects. Intuitively, the model includes no unnecessary syntactic detail and describes a class in terms of the external behaviour of its objects.

Chapter 6 discusses the concept of behavioural compatibility as a basis for refinement in Object-Z. As well as a general definition of behavioural compatibility, two weaker notions of behavioural compatibility are defined. *Observational compatibility* is relevant when an active object is placed within a passive environment and *operational compatibility* when a passive object is placed in an active environment. Rules for maintaining each type of behavioural compatibility through inheritance are also presented.

The thesis is concluded in Chapter 7 by providing a summary of the major points, evaluating the contributions with respect to related work and indicating future research directions.

Chapter 2

Classes

> *"We live in a world of things, and our only connection with them is that we know how to manipulate or to consume them."*
>
> — Erich Fromm
> *The Sane Society*, 1955.

Z is a state-based formal specification language based on the established mathematics of set theory and first-order predicate logic[1]. It has been developed, over the past ten years, from the work of Abrial *et al.*[2] by the Programming Research Group at Oxford University (e.g. see [106, 83, 56, 107, 108]) and has been used to specify a wide range of systems including transaction processing systems[86] and communications protocols[40, 59].

A specification in Z typically consists of a number of state and operation schemas. A state schema groups together variables and defines the relationship that holds between their values. An operation schema defines the relationship between the 'before' and 'after' values of variables belonging to one or more state schemas. Inferring the operation schemas that may affect a particular state schema requires examining the signatures of all schemas in the specification. In a large specification, containing numerous state and operation schemas, this task may prove impracticable.

This problem can be overcome by extending Z to facilitate an object-oriented approach to formal specification. A fundamental idea of object orientation is that the state of an object may only be changed by operations within its class. By adopting the notion of class, therefore, the relationship between state and operation schemas can be made explicit. Furthermore, the enhanced structuring enables the specifier to incorporate elements of design into a specification.

[1] A familiarity with Z will be assumed throughout this thesis. A glossary of the Z notation used is included in Appendix B.

Object-Z is an extension of Z in which the existing syntax and semantics of Z are retained and new constructs are introduced to facilitate specification in an object-oriented style. The major extension in Object-Z is the *class schema* which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are objects. Section 2.1 presents the syntax of a basic class schema. This syntax will be extended in later chapters as new features are introduced. Section 2.2 presents a set-theoretic model of classes which could be used as the basis of a full formal semantics of Object-Z. This model is used to explain the meaning of object instantiation in Section 2.3.

2.1 Syntax of Classes

The objective of this section is to provide an informal description of the syntax of a basic class schema in Object-Z. A formal description of this syntax can be found in Appendix A.

An Object-Z class schema, often referred to simply as a class, is represented syntactically as a named box with zero or more generic parameters. In this box there may be local type and constant definitions, at most one state and associated initial state schema and zero or more operations. A class may also include the names of inherited classes (introduced in Chapter 3) and history invariants for capturing liveness properties (introduced in Chapter 4). For this chapter, the basic structure of a class is as follows.

ClassName[*generic parameters*]

type definitions
constant definitions
state schema
initial state schema
operations

2.1.1 Class features

The type and constant definitions within an Object-Z class have the same syntax as global type and constant definitions in Z. Their scope, however, is limited to the class in which they are declared. A constant is associated with a fixed value which cannot be changed by any of the operations of the class. However, the value of constants may differ for different instantiations, i.e. objects, of the class.

The state schema is like a Z state schema except that it has no name associated with it. The declarations of the state schema are referred to as the *state variables* and the predicate as the *state invariant*. The state invariant restricts the possible values of not only the

state variables but also the constants. The state variables and constants collectively define the class attributes.

The initial state schema is distinguished by the name *Init*. Since the state and the initial state schema are encapsulated within a class, there is no need to explicitly include the state variables and state invariant in the initial state schema. Instead, they are included implicitly.

Similarly, the state variables and state invariant in both primed and unprimed form are implicitly included in each of the operations of the class. The state invariant is, therefore, true at all times, i.e. in each possible initial state and before and after each operation.

An operation is either an operation schema or a schema expression involving existing class operations and schema operators similar to those in Z. The operation schemas within an Object-Z class differ from Z operation schemas in that they have, in addition to the declaration and predicate parts, a Δ-*list*. The Δ-list of an operation contains a subset of the state variables. The understanding is that when the operation is *applied* to an object of the class, those variables not in the Δ-list remain unchanged. (Application of class operations to objects is discussed in Section 2.3.) When two or more operations are combined in a schema expression, the Δ-list of the resulting operation unites all of the variables in the Δ-lists of the constituent operations.

As an example, consider the following class C.

C

$c : \mathbb{N}$

$x, y : \mathbb{N}$

$x \leqslant y + c$

Init

$x = y = 0$

IncX

$\Delta(x)$

$x' = x + 1$

IncY

$\Delta(y)$

$y' = y + 1$

$IncBoth \mathrel{\widehat=} IncX \land IncY$

$IncEither \mathrel{\widehat=} IncX \lor IncY$

The class has a constant c and two state variables x and y. The state invariant stipulates that the value of x cannot exceed the value of y by more than c.

Initially, both x and y have the value zero. The initial state schema could be expanded, by explicitly including the state variables and state invariant, to yield the semantically identical schema shown below.

INIT

$$x, y : \mathbb{N}$$

$$x \leqslant y + c$$
$$x = y = 0$$

Two of the class operations, *IncX* and *IncY*, are defined by operation schemas and the other two, *IncBoth* and *IncEither*, by schema expressions involving *IncX* and *IncY*. The operations *IncX* and *IncY* increment the values of the state variables x and y respectively. They could be expanded to yield the following semantically identical schemas.

IncX

$$\Delta(x)$$
$$x, x', y, y' : \mathbb{N}$$

$$x \leqslant y + c$$
$$x' \leqslant y' + c$$
$$x' = x + 1$$

IncY

$$\Delta(y)$$
$$x, x', y, y' : \mathbb{N}$$

$$x \leqslant y + c$$
$$x' \leqslant y' + c$$
$$y' = y + 1$$

On application of operation *IncX* to an object of class C the value of y will be unchanged, i.e. $y' = y$, as it is not included in the operation's Δ-list. Similarly, on application of operation *IncY* to an object of class C the value of x will be unchanged. This property is referred to as *late binding* as the variables are bound to a particular value only on application of the operation and not on its declaration.

The late binding of Δ-lists allows operations within a class to be combined with maximum flexibility. For example, the operation *IncBoth*, which increments the values of x and y simultaneously, is defined as the conjunction of the operations *IncX* and *IncY*. It is semantically identical to the following operation schema.

IncBoth

$$\Delta(x, y)$$

$$x' = x + 1$$
$$y' = y + 1$$

Without the late-binding property of Δ-lists, the expanded version of *IncX* would need to include the predicate $y' = y$. Similarly, the expanded version of *IncY* would need

to include the predicate $x' = x$. Since *IncX* includes a predicate to increment x and *IncY* a predicate to increment y, conjoining the expanded schemas would result in an operation whose precondition and postcondition were always false. Hence, the operation which increments both variables could not be defined as a schema expression conjoining *IncX* and *IncY*.

The late binding property of Δ-lists must be taken into account when combining operations to ensure the resulting operation will behave as desired. For example, consider the operation *IncEither* which is defined as the disjunction of the operations *IncX* and *IncY*. It is semantically identical to the following operation schema.

IncEither

$\Delta(x, y)$

$(x' = x + 1$
$\lor$
$y' = y + 1)$

This operation may increment x only, y only or both x and y. In the first and second cases, the variable which is not incremented may take on any value consistent with its type and the state invariant. This is because it appears in the operation's Δ-list but is not constrained by the operation's predicate.

If, however, the intention is to specify an operation which increments one variable leaving the other unchanged then the following approach is appropriate. An extended version of *IncX*, *IncJustX*, is introduced which adds the constraint that y does not change, i.e. the Δ-list of *IncX* is widened to include y and $y' = y$ is added to its predicate. Similarly, an extended version of *IncY*, *IncJustY*, is introduced.

IncJustX

$\Delta(y)$
IncX

$y' = y$

IncJustY

$\Delta(x)$
IncY

$x' = x$

The desired operation can then be defined as follows.

$$IncExactlyOne \mathrel{\widehat=} IncJustX \lor IncJustY$$

It is semantically identical to the following operation schema.

IncExactlyOne

$\Delta(x, y)$

$((x' = x + 1 \land y' = y)$
$\lor$
$(y' = y + 1 \land x' = x))$

An operation which increments either one or both variables could also be defined by a schema expression involving the operations *IncExactlyOne* and *IncBoth*.

$$IncOneOrBoth \mathrel{\widehat=} IncExactlyOne \lor IncBoth$$

2.1.2 Parameterised classes

Classes in Object-Z may have generic parameters corresponding to arbitrary types. The generic parameters are listed after the class name in square brackets and may be used in the definition of local types and constants as well as in the declaration of variables in the state and operation schemas. Generic parameters reduce the need to specify almost identical classes and are particularly useful for specifying classes representing data structures such as queues, stacks and arrays. For example, consider the following Object-Z specification of a generic queue class. The queue is modelled as a sequence of items which is initially empty. Operations are provided to allow items to join or leave the queue on a first-in/first-out basis.

Queue[T]

$items : \text{seq } T$

Init

$items = \langle\,\rangle$

Join

$\Delta(items)$
$item? : T$

$items' = items \frown \langle item? \rangle$

Leave

$\Delta(items)$
$item! : T$

$items \neq \langle\,\rangle$
$items = \langle item! \rangle \frown items'$

The class has a single generic parameter T corresponding to the type of the items in the queue. It is used in the declaration of the state variable *items* as well as in the declaration of the input and output variables *item*? and *item*! in the operations *Join* and *Leave* respectively.

T is called a *formal generic parameter*. When an object of class *Queue*[T] is declared, the parameter T must be instantiated by an *actual generic parameter*. An example of

instantiation of a formal generic parameter with an actual generic parameter is given in Section 2.3.

An actual generic parameter may be any type, including a class type. If the object is declared within another generic class then the actual generic parameter may also be a formal generic parameter of that class. The expressions involving variables of a generic type must be applicable to all possible types so that the substitution of any type for the formal generic parameter is possible.

2.2 Semantics of Classes

A full formal semantics of Object-Z, i.e. a mapping from an abstract syntax to some semantic domain, is beyond the scope of this thesis[2]. Instead, a formal model of classes is presented which could form the basis for such a semantics.

To enable classes in Object-Z to be used as types, the meaning of a class is taken to be a set of values. Each value corresponds to a potential object of the class at some stage of its evolution. The value chosen to represent an object is the sequence of states the object has passed through together with the corresponding sequence of events the object has undergone. This value is referred to as the *history* of the object.

As a preliminary, a model based upon the syntactic structure of classes is presented in Section 2.2.1. This model is used to derive the history model of classes in Section 2.2.2. Both models are defined using the specification language Z.

2.2.1 Structural model

Structurally, a class in Object-Z consists of a set of attributes and a set of operations which act upon those attributes. The set of attributes includes, as well as all local constants and state variables of the class, any global constants to which the class may refer. Each operation has a set of parameters for the purpose of input and output.

Attributes and operations within a class are given unique names, or *identifiers*. Also, parameters within a particular operation are identified uniquely with respect to each other and with respect to the attributes and operations of the class.

Let Id denote the set of all possible identifiers. In Object-Z, this would be the set of all strings of alphanumerics, underscores and symbols excluding certain reserved symbols. Such details will not be elaborated upon formally here.

[2]Preliminary work on a full formal semantics for Object-Z can be found in [39].

The signature of a class can be defined in terms of its attributes, operations and operation parameters as follows.

$$
\begin{array}{|l}
\textit{ClassSig} \\
\hline
attr : \mathbb{P}\, Id \\
ops : \mathbb{P}\, Id \\
op\_params : Id \pfun \mathbb{P}\, Id \\
\hline
attr \cap ops = \varnothing \\
\mathrm{dom}\ op\_params = ops \\
\forall\, o : ops \bullet op\_params(o) \cap (attr \cup ops) = \varnothing \\
\hline
\end{array}
$$

Let *Value* denote the set of all possible values that could be assigned to any identifier of any type. A *state* is an assignment of values to a set of identifiers representing attributes. It can be defined as a finite partial function from identifiers to values as follows.

$$State == Id \fpfun Value$$

An *event* is an occurrence of an operation. It can be defined as a tuple consisting of the operation's name and a finite partial function defining the values of the operation's parameters.

$$Event == Id \times (Id \fpfun Value)$$

The auxiliary functions *op* and *params* are defined to enable access to an event's associated operation name and parameter values respectively.

$$
\begin{array}{|l}
op : Event \to Id \\
params : Event \to (Id \fpfun Value) \\
\hline
\forall\, e : Event \bullet e = (op(e), params(e)) \\
\hline
\end{array}
$$

The structural model of a class defines a set of objects in terms of the states they can be in and the events they can undergo. It extends the definition of the signature of the class to include the following.

states - the set of possible states of an object of the class. This includes those states which are composed of the attributes of the class (including any global constants) and which satisfy the state invariant of the class.

initial - the set of possible initial states of an object of the class. This includes those states from the set of possible states of an object of the class which satisfy the predicate of the initial state schema of the class.

trans - a function from the set of possible events an object of the class may undergo to the associated set of state transitions. This includes, for each event, those pairs of states from the set of possible states of an object of the class which satisfy the precondition and resulting postcondition of the operation associated with the event. The interpretation of operations in an Object-Z class differs from that in Z in that an operation cannot occur when its precondition is not enabled. In Z, the operation would be able to occur but the outcome would be unspecified.

ClassStruct

$$
\begin{array}{l}
ClassSig \\
states : \mathbb{P}\, State \\
initial : \mathbb{P}\, State \\
trans : Event \pfun (State \rel State) \\
\hline
\forall\, s : states \bullet \operatorname{dom} s = attr \\
initial \subseteq states \\
\forall\, e : \operatorname{dom} trans \bullet \\
\quad op(e) \in ops \\
\quad \operatorname{dom} params(e) = op\_params(op(e)) \\
\quad trans(e) \subseteq states \rel states
\end{array}
$$

The predicate of the schema *ClassStruct* relates the possible states, initial states and events of objects of the class to the class signature.

The schema *ClassStruct* effectively defines a state transition system. The same information can, therefore, also be represented graphically as a state transition diagram. As an example, consider the following Object-Z specification of a simple vending machine.

VendingMachine

$credit : \{0, 50, 100, 150\}$

Init

$credit = 0$

Coin

$\Delta(credit)$
$coin? : \{50, 100\}$

$credit < 100$
$credit' = credit + coin?$

Choc

$\Delta(credit)$
$change! : \{0, 50\}$

$credit \geqslant 100$
$change! = credit - 100$
$credit' = 0$

The vending machine allows a customer to purchase a chocolate for one dollar by inserting either one dollar or 50 cent coins. The state variable *credit* denotes the amount of money inserted by a customer. Initially, the credit is zero. The operation *Coin* represents the customer inserting a coin, denoted by the input parameter *coin*?, whose value is added to *credit*. The precondition of *Coin* prevents a customer inserting another coin when the credit already exceeds one dollar. The operation *Choc* represents the customer receiving a chocolate. The precondition allows this operation to occur only when the credit is at least one dollar. The customer also receives change, denoted by the output parameter *change*!, and the credit is returned to zero.

The sets *attr*, *ops* and the function *op_params* for this class are as follows.

$$attr = \{\text{`}credit\text{'}\}$$

$$ops = \{\text{`}Coin\text{'}, \text{`}Choc\text{'}\}$$

$$op\_params = \{\text{`}Coin\text{'} \mapsto \{\text{`}coin?\text{'}\}, \text{`}Choc\text{'} \mapsto \{\text{`}change!\text{'}\}\}$$

The notation '*credit*' denotes the name of attribute *credit* as opposed to its semantic value. Similarly, the notations '*Coin*', '*Choc*', '*coin*?' and '*change*!' denote the names of the corresponding operations and parameters.

The sets *states*, *initial* and the function *trans* for the class *VendingMachine* are as follows.

$$states = \{\{\text{`}credit\text{'} \mapsto 0\}, \{\text{`}credit\text{'} \mapsto 50\}, \{\text{`}credit\text{'} \mapsto 100\}, \{\text{`}credit\text{'} \mapsto 150\}\}$$

$$initial = \{\{\text{`}credit\text{'} \mapsto 0\}\}$$

$$\begin{aligned} trans = \{&(\text{`}Coin\text{'}, \{\text{`}coin?\text{'} \mapsto 50\}) \mapsto \{(\{\text{`}credit\text{'} \mapsto 0\}, \{\text{`}credit\text{'} \mapsto 50\}),\\ &\qquad (\{\text{`}credit\text{'} \mapsto 50\}, \{\text{`}credit\text{'} \mapsto 100\})\},\\ &(\text{`}Coin\text{'}, \{\text{`}coin?\text{'} \mapsto 100\}) \mapsto \{(\{\text{`}credit\text{'} \mapsto 0\}, \{\text{`}credit\text{'} \mapsto 100\}),\\ &\qquad (\{\text{`}credit\text{'} \mapsto 50\}, \{\text{`}credit\text{'} \mapsto 150\})\},\\ &(\text{`}Choc\text{'}, \{\text{`}change!\text{'} \mapsto 0\}) \mapsto \{(\{\text{`}credit\text{'} \mapsto 100\}, \{\text{`}credit\text{'} \mapsto 0\})\},\\ &(\text{`}Choc\text{'}, \{\text{`}change!\text{'} \mapsto 50\}) \mapsto \{(\{\text{`}credit\text{'} \mapsto 150\}, \{\text{`}credit\text{'} \mapsto 0\})\}\} \end{aligned}$$

An equivalent state transition diagram is shown in Figure 2.1. Circles represent states and labelled arcs represent events. A circle with an unlabelled arc entering it is an initial state.

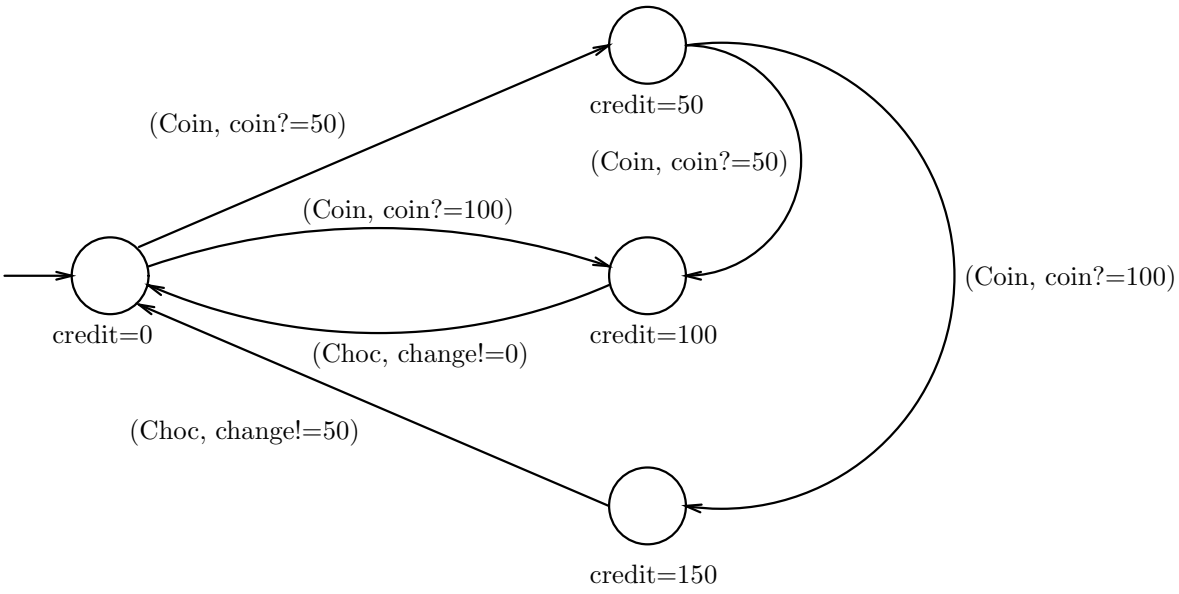


Figure 2.1: State transition diagram of the *VendingMachine* class.

2.2.2 History model

The history model defines the *type* of a class in Object-Z as opposed to its syntactic structure. This type is defined to be the set of histories that objects of the class can undergo. Intuitively, the history of an object represents the sequence of states the object has passed through together with the sequence of operations it has undergone. It can be modelled as a non-empty sequence of states ss, where each state assigns values to a common set of identifiers, and a sequence of events es such that the number of states in ss is one more than the number of events in es unless ss is infinite in which case es is

also infinite. The relationship between the sequence of states *ss* and the corresponding sequence of events *es* of a history can be represented by a state transition diagram as shown in Figure 2.2.

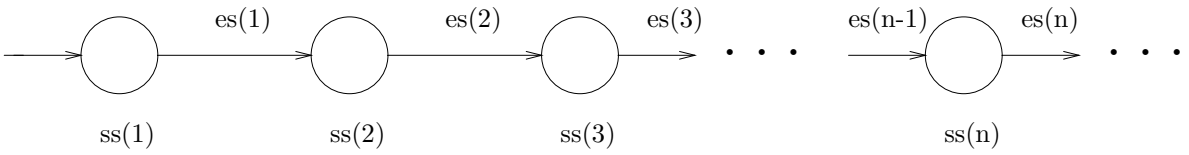


Figure 2.2: State transition diagram representation of a history.

Building on the definitions in Section 2.2.1, a history is defined as follows.

History

$$
\begin{array}{l}
states : \text{seq}_\infty\ State \\
events : \text{seq}_\infty\ Event \\
\hline
states \neq \langle\,\rangle \\
\forall\, i, j : \text{dom}\ states \bullet \text{dom}\ states(i) = \text{dom}\ states(j) \\
\forall\, i : \mathbb{N}_1 \bullet i \in \text{dom}\ events \Leftrightarrow i + 1 \in \text{dom}\ states
\end{array}
$$

Any history whose sequences of states and events are prefixes of or equal to those of another history is referred to as a *pre-history* of that history. Intuitively, a pre-history of an object's history represents the history of that object at some stage of its evolution.

$$
\begin{array}{l}
prehist : History \rightarrow \mathbb{P}\ History \\
\hline
\forall\, h : History \bullet \\
\quad prehist(h) = \{ph : History \mid ph.states \subseteq h.states \land ph.events \subseteq h.events\}
\end{array}
$$

The set of histories representing a class can be derived from the structural model of the class using the function $\mathcal{H}$ defined below.

$$
\begin{array}{l}
\mathcal{H} : ClassStruct \rightarrow \mathbb{P}\ History \\
\hline
\forall\, c : ClassStruct \bullet \\
\quad \mathcal{H}(c) = \{h : History \mid \\
\qquad\qquad h.states(1) \in c.initial \land \\
\qquad\qquad \forall\, i : \text{dom}\ h.events \bullet \\
\qquad\qquad\quad h.events(i) \in \text{dom}\ c.trans \land \\
\qquad\qquad\quad (h.states(i), h.states(i+1)) \in c.trans(h.events(i))\}
\end{array}
$$

The first state in the sequence of states of any history of a class is an initial state of the class and each pair of consecutive states is a possible state transition of the corresponding event in the sequence of events.

An important property of the set of histories of a class is that it is *pre-history closed*, i.e. given any history in the set, all pre-histories of that history are also in the set. This is necessary as any pre-history of an object's history is the history of that object at some earlier stage in its evolution and hence represents a possible history of the object's class.

2.3 Object Instantiation

A fundamental concept of object orientation is that objects may be composed of other objects. In effect, the attributes of an object may themselves be objects.

Adopting the history model of classes, a class in Object-Z may be used as a type and incorporated into the Z type system. An object may then be declared as an instance of a class. For example, if C is a class then the declaration $c : C$ declares the object c to be of class C. The semantic representation of such an object is a history from the set of histories representing its class. That is, if $\mathsf{C} : \mathit{ClassStruct}$ is the structural model of the class C then $c \in \mathcal{H}(\mathsf{C})$.

Adopting the point of view that the state of an object is hidden, its attributes may not be accessed directly. An object may only be accessed through the procedural interface defined by its class. If the value of an attribute is required in the environment of an object then an operation must be included in the object's class to return this value.

In this section, the use of objects to specify systems in Object-Z is illustrated through two examples. The first example, a simple communication channel, involves instantiation of classes with generic parameters and introduces the use of the dot notation for initialising and applying operations to objects. The second example, a simple card game, looks at specifying aggregates of objects and the concept of object identity in Object-Z.

2.3.1 Composite objects

An object can be specified in terms of other objects. Although the functionality of such an object can usually be specified without modelling its components, it is important to be able to specify the components to enable refinement to a low level of abstraction and to enhance clarity when the system is not easily described except in terms of its components.

The class of such an object has objects of other classes as state variables. When declaring an object to be of a particular class, any formal generic parameter of that class must be replaced by an actual generic parameter, i.e. a type or a formal generic parameter of the class in which the declaration occurs.

As an example, consider a simple communication channel consisting of two queues of messages joined together so that the output from the first queue becomes the input for

the second queue. Using the class $Queue[T]$ specified in Section 2.1.2, the class of this channel may be specified as follows.

Let MSG denote the set of all possible messages that may be sent along the channel.

Channel

$q_1, q_2 : Queue[MSG]$

Init

$q_1.\mathit{Init}$
$q_2.\mathit{Init}$

$Join \mathrel{\widehat=} q_1.Join$

$Transfer \mathrel{\widehat=} q_1.Leave \parallel q_2.Join$

$Leave \mathrel{\widehat=} q_2.Leave$

The class *Channel* has two state variables, q_1 and q_2, representing objects of the class $Queue[T]$ with the formal generic parameter T instantiated with the type MSG.

Dot notation

The class *Channel* illustrates the use of the dot notation to initialise and apply operations to objects. Semantically, the notation $q_1.\mathit{Init}$ is identical to the schema which states that the object q_1 has not undergone any events.

$q_1.\mathit{Init}$

$q_1.events = \langle\,\rangle$

It can be deduced, from the definition of $\mathcal{H}$, that the state of q_1, i.e. the final state in q_1's state sequence, satisfies the predicate of the *Init* schema of its class. That is, the predicate $items = \langle\,\rangle$ is true. Similar meaning can be given to the notation $q_2.\mathit{Init}$.

The notation $q_1.Join$ can also be represented semantically as a schema which states that the object q_1 undergoes an event associated with the operation *Join* provided the sequence of events q_1 has undergone is finite. The condition of finiteness is required as only appending to a finite sequence is defined. The input variable *item*? declared in *Join* is included in the signature of this schema. In general, all input and output variables declared in the corresponding operation's signature would be similarly included.

$q_1.Join$

$\Delta(q_1)$
$item? : MSG$

$q_1.events \in \operatorname{seq} Event$
$front\ q_1'.states = q_1.states$
$q_1'.events = q_1.events \frown \langle(\text{`}Join\text{'}, \{\text{`}item?\text{'} \mapsto item?\})\rangle$

It can be deduced that if $q_1.Join$ occurs then the state of q_1 before the operation satisfies the precondition of the operation *Join* and the state of q_1 after the operation is related to the state before by a state transition defined by *Join*. Similar meanings can be given to the notations $q_1.Leave$, $q_2.Join$ and $q_2.Leave$.

The above schema expansions of expressions involving the dot notation are given only to clarify the meaning of such expressions with respect to the history model of Section 2.2.2. In practice, expressions which equate an object to a history are not allowed in Object-Z specifications as they allow objects to be used in ways other than specified by their class. Expressions such as $a = b$ and $a \in A$, where a and b are objects and A is a set of objects, are also not allowed for similar reasons discussed in Chapter 7.

An object may only be referred to in the Δ-list of an operation, in expressions involving the dot notation as detailed in this and the following section or in history invariants as detailed in Chapter 4.

Parallel operator

The parallel operator $\|$ used in the operation *Transfer* of class *Channel* is a binary operator introduced into Object-Z to allow specification of inter-object communication. The operator conjoins two operations but also identifies and equates input variables in either schema with output variables in the other schema having the same type and basename, i.e. apart from the '?' or '!'. These identified input and output variables are hidden in the resulting operation, i.e. they are not available in the environment of objects of the class. The operation *Transfer* is semantically identical to the following operation schema.

Transfer

$\Delta(q_1, q_2)$

$\exists\, item?, item! : MSG \bullet$
$\quad item? = item!$
$\quad q_1.Leave$
$\quad q_2.Join$

The parallel operator may be compared with the concurrency operators used to capture synchronisation in process algebras such as CSP[61]. The main difference is in the level

at which the operator is applied. The parallel operator of Object-Z is applied between operations, or events, whereas the concurrency operator of a process algebra is applied between the specifications of processes, or objects. The process algebra approach leads to specifications which are more concise, but also more restrictive in that operations which synchronise have to have the same name. The Object-Z approach not only allows more flexibility when naming operations, but also allows objects to be declared with other non-object state variables allowing additional state information to be captured in the system specification. The ability to distribute state information between the system and component specifications aids in handling complexity in large systems as detailed in [104].

2.3.2 Aggregates of objects

Often systems are composed of aggregates of objects of the same class. For example, a telephone system consists of a collection of telephones and exchanges and a multiprocessor system consists of a collection of processors.

Each object within such an aggregate is assumed to have its own identity which persists through time. This enables it to be distinguished from any other object in the aggregate even if the other object has undergone exactly the same events and hence has the same history.

In the class *Channel* the objects q_1 and q_2 were uniquely identified by their names. In aggregates where the number of objects is large or arbitrary, however, it may not be practical or possible to uniquely instantiate each object as a state variable. Instead, objects may be instantiated as members of a set with the notion of object identity captured by specifying a mapping from a set of identifiers into this set. To illustrate this idea, consider the following specification of a simple card game.

Card Game Example

The game is played by an arbitrary collection of players. The game begins with each player having the same even number of cards from a given pack. A move consists of a player selecting another player and offering the selected player some cards. The selected player must reply by accepting the cards and offering, from his or her original hand, the same number of cards in response. At any time, players can discard pairs of cards in their hand with the same face value. The game continues as long as there are two or more players still holding cards.

Let *Card* denote the set of all possible cards. A pack of cards may contain more than one of the same card. The function $value : Card \rightarrow \mathbb{N}$ gives the face value of each card.

Let $Even == \{n : \mathbb{N} \mid \exists\, m : \mathbb{N} \bullet n = 2m\}$ be the set of all even numbers and $dealt : Even$ be a constant denoting the number of cards initially held by each player. Using these definitions, the class of a player can be specified as follows.

Player

$$hand : \mathrm{seq}\ Card$$

Init

$$\#hand = dealt$$

SelectPartner

$$\Delta(hand)$$
$$offer!, respond? : \mathrm{seq}\ Card$$

$$\#respond? = \#offer!$$
$$\exists f : \mathrm{dom}\ offer! \rightarrowtail \mathrm{dom}\ hand \bullet$$
$$\quad \forall i : \mathrm{dom}\ f \bullet offer!(i) = hand(f(i))$$
$$\quad hand' = squash(\mathrm{ran}\ f \lhd hand) \frown respond?$$

SelectedAsPartner

$$\Delta(hand)$$
$$offer?, respond! : \mathrm{seq}\ Card$$

$$\#respond! = \#offer?$$
$$\exists f : \mathrm{dom}\ respond! \rightarrowtail \mathrm{dom}\ hand \bullet$$
$$\quad \forall i : \mathrm{dom}\ f \bullet respond!(i) = hand(f(i))$$
$$\quad hand' = squash(\mathrm{ran}\ f \lhd hand) \frown offer?$$

DiscardPairs

$$\Delta(hand)$$

$$\exists discard : \mathrm{seq}\ Card \bullet$$
$$\quad \exists f : \mathrm{dom}\ discard \rightarrowtail \mathrm{dom}\ hand \bullet$$
$$\quad\quad \forall i : \mathrm{dom}\ f \bullet discard(i) = hand(f(i))$$
$$\quad\quad \forall n : \mathbb{N} \bullet \#(discard \fatsemi value \rhd \{n\}) \in Even$$
$$\quad\quad hand' = squash(\mathrm{ran}\ f \lhd hand)$$

The class has a single state variable *hand* denoting the player's hand. Initially, the player holds an even number of cards corresponding to the constant *dealt*.

The operation *SelectPartner* corresponds to the player selecting a partner and swapping some cards, the operation *SelectedAsPartner* corresponds to the player being selected as a partner and swapping some cards, and the operation *DiscardPairs* corresponds to the player discarding some or all matching pairs of cards.

The class denoting the game can be specified as follows.

Game

$[Name]$

$players : Name \pfun Player$

INIT

$\forall\, n : \mathrm{dom}\, players \bullet (players, n).INIT$

$\Phi SelectPlayer$

$\Delta(players)$
$n : Name$

$n \in \mathrm{dom}\, players$
$\{n\} \ndres players' = \{n\} \ndres players$

$\Phi SelectPlayers$

$\Delta(players)$
$n_1, n_2 : Name$

$n_1 \neq n_2$
$\{n_1, n_2\} \subseteq \mathrm{dom}\, players$
$\{n_1, n_2\} \ndres players' = \{n_1, n_2\} \ndres players$

$Discard \mathrel{\widehat=} \Phi SelectPlayer \bullet (players, n).DiscardPairs$

$Swap \mathrel{\widehat=} \Phi SelectPlayers \bullet$
$((players, n_1).SelectPartner \parallel (players, n_2).SelectedAsPartner)$

The class *Game* has a local type *Name* denoting the set of all possible names of players of the game. The state variable *players* is a partial function which associates each object of class *Player* in its range with a unique identifier from the set of names.

The class illustrates an alternative usage of the dot notation for objects in the range of a function. The notation $(players, n).INIT$ can only be interpreted in the scope of the declaration of a variable n representing a possible domain value of the function *players*. Given $n : Name$, the notation $(players, n).INIT$ is semantically identical to the following schema.

$(players, n).INIT$

$n : Name$

$n \in \mathrm{dom}\, players$
$players(n).events = \langle\, \rangle$

Framing schemas

The class *Game* has two framing schemas identified by names beginning with the symbol Φ. These schemas don't correspond to actual operations but can be combined with other schemas to define operations. They are generally used to change one, or a limited number, of objects in an aggregate leaving the rest unchanged.

For example, the framing schema $\Phi SelectPlayer$ specifies that the single player identified by the name n is updated and all other players remain unchanged. The schema does not, however, specify how the selected player is updated. Similarly, the framing schema $\Phi SelectPlayers$ specifies that the players identified by the names n_1 and n_2 are updated and all other players remain unchanged.

These framing schemas are used in the definitions of the operations *Discard* and *Swap* to specify a player discarding matching pairs of cards and two players exchanging cards respectively.

Nesting operator

The definitions of the operations *Discard* and *Swap* in class *Game* also involve the use of the nesting operator $\bullet$. This operator is introduced into Object-Z to enable schemas to be defined within a scope which includes the declarations of another schema.

For example, the notation $\Phi SelectPlayer \bullet (players, n).DiscardPairs$ allows variables declared in the signature of the framing schema $\Phi SelectPlayer$ to be used when interpreting $(players, n).DiscardPairs$. This is necessary as the variable n is not declared in the state schema of class *Game*.

Given the declarations $n : Name$ from the signature of $\Phi SelectPlayer$, however, the notation $(players, n).Discard$ can be represented semantically as follows.

$(players, n).DiscardPairs$

$\Delta(players)$
$n : Name$

$n \in \operatorname{dom} players \cap \operatorname{dom} players'$
$players(n).events \in \operatorname{seq} Event$
$front\ players'(n).states = players(n).states$
$players'(n).events = players(n).events \frown \langle(\text{`}DiscardPairs\text{'}, \varnothing)\rangle$

The operation *Discard* in class *Game* can be represented semantically as the conjunction of this schema with $\Phi SelectPlayer$.

Discard

$\Delta(players)$
$n : Name$

$n \in \operatorname{dom} players \cap \operatorname{dom} players'$
$players(n).events \in \operatorname{seq} Event$
$front\ players'(n).states = players(n).states$
$players'(n).events = players(n).events \frown \langle(\text{`}DiscardPairs\text{'}, \varnothing)\rangle$
$\{n\} \lhd players' = \{n\} \lhd players$

The operation *Swap* in class *Game* can be interpreted in a similar fashion.

The parallel and nesting operators constitute the only additional schema operators introduced into Object-Z. All existing Z schema operators are also available within a class with the exception of the schema composition operator $\fatsemi$. The reasons for the exclusion of this operator are discussed in Chapter 7.

Chapter 3

Inheritance

"Begin with another's to end with your own."

— Baltasar Gracián
The Art of Worldly Wisdom, 1647.

Classes provide a means of modular decomposition by encapsulating the state information and operations of the constituent objects of a system. Such objects are declared as instances of a class type and, hence, a class may be reused to define a number of objects possessing the same state information and operations. The full benefits of reusability of classes can only be realised, however, when classes may also be used in the definition of other classes.

Inheritance allows a class to be defined as an extension or specialisation of another class by a process of incremental modification. In this way, the repetition of structure between class definitions can be avoided. Inheritance may, in general, involve the addition, cancellation and renaming of attributes and operations as well as the redefinition of operations.

In many object-oriented languages, inheritance is restricted so that the classes within an inheritance hierarchy can be used polymorphically. That is, an object declared to be of a particular class may be assigned to an object of that class or any class derived from that class by inheritance. This concept is particularly useful in object-oriented specification languages as it allows an object to be defined as a member of a set of classes thus providing greater freedom to the system implementor.

Object-Z supports both single and multiple inheritance and allows inheritance hierarchies to be used polymorphically. Section 3.1 introduces inheritance in Object-Z and Section 3.2 discusses renaming and redefinition. Polymorphism in Object-Z is discussed in Section 3.3. Inheritance in Object-Z is not restricted to maintain compatibility between a class and its subclasses and, hence, it is the responsibility of the specifier to ensure compatibility exists when an inheritance hierarchy is used polymorphically. Rules for maintaining signature

compatibility through inheritance are presented in Section 3.3. Rules for maintaining the stronger notion of behavioural compatibility are presented in Chapter 6.

3.1 Introduction to Inheritance

Inheritance is achieved syntactically in Object-Z by including the names of the inherited classes within the inheriting class. Informally, the syntactic structure of a class which inherits one or more other classes is an extension of the syntactic structure of a class defined in Section 2.1 as follows.

ClassName[*generic parameters*]

inherited classes
type definitions
constant definitions
state schema
initial state schema
operations

3.1.1 Meaning of inheritance

When a class in Object-Z inherits another class, the type, constant and schema definitions in the inherited class are *merged* with those declared explicitly in the inheriting class. Any type or constant names which occur in both classes are semantically identified and must, therefore, be associated with identical types. The state schemas, initial state schemas and any operations with the same name are conjoined. (Operations defined as schema expressions are first expanded to a semantically identical schema as discussed in Section 3.1.3.) To ensure merging occurs without conflict, any variable names which occur in the state schema or a common-named operation schema in both classes must be associated with identical types.

Person/Student example

As a preliminary example, consider the following classes *Person* and *Student* based on the Eiffel classes *PERSON* and *STUDENT* defined in Section 1.2.2. In this example, there are no type, constant or operation names common to both the inherited and the inheriting class and no common-named variables in the state schemas of the classes. Inheritance, therefore, corresponds simply to including the attributes and operations of the inherited class in the inheriting class.

Let *Name* denote the set of all names. The class *Person* is specified as follows.

Person

$name : Name$

$age : \mathbb{N}$

Birthday

$\Delta(age)$

$age' = age + 1$

The class *Person* has a constant *name* denoting the name of the person and a state variable *age* denoting the person's age. A person's age is initially undefined[1] and may be incremented by the operation *Birthday*.

The class *Student* is an extension of the class *Person* which includes information about the number of subjects studied and operations corresponding to enrolling in and completing subjects.

Student

Person

$nb\_subjects : \mathbb{N}$

Enrol

$\Delta(nb\_subjects)$

$nb\_subjects < 8$
$nb\_subjects' = nb\_subjects + 1$

Complete

$\Delta(nb\_subjects)$

$nb\_subjects > 0$
$nb\_subjects' = nb\_subjects - 1$

The class *Student* inherits the constant *name*, the state variable *age* and the operation *Birthday* from *Person*. It also includes, explicitly, the state variable *nb_subjects* denoting the number of subjects studied and the operations *Enrol* and *Complete* corresponding to enrolling in a new subject and completing a subject respectively. The operation *Enrol* has a precondition restricting a student from enrolling in more than 8 subjects and the

[1]The class may be thought of as defining a set of records which may be created for people of various ages.

operation *Complete* has a precondition stating that the number of subjects studied is greater than zero.

The class *Student* could be expanded, by explicitly including the inherited attributes and operations of the class *Person*, to yield the following semantically identical class.

Student

$name : Name$

$age : \mathbb{N}$
$nb\_subjects : \mathbb{N}$

Birthday

$\Delta(age)$

$age' = age + 1$

Enrol

$\Delta(nb\_subjects)$

$nb\_subjects < 8$
$nb\_subjects' = nb\_subjects + 1$

Complete

$\Delta(nb\_subjects)$

$nb\_subjects > 0$
$nb\_subjects' = nb\_subjects - 1$

As well as addition of attributes and operations, inheritance in Object-Z also allows addition of constraints to the state invariant, to the initial condition or to the predicate of an operation schema. This idea is illustrated in the following section by the 'shapes' example. This example was inspired by the tutorial examples for the object-oriented programming languages Eiffel and C++ in [80] and [112] respectively.

3.1.2 The 'shapes' example

There are various approaches to modelling shapes in a specification language such as Object-Z. For example, the set of all points which make up the boundary of an 'outline' shape, or the set of all points which make up a 'solid' shape could be specified. Such specifications allow operations on shapes to be defined at a high-level, but add complexities that are not warranted here. Instead, a simple model of shapes is adopted to facilitate discussion of the issues of inheritance.

A class hierarchy of geometric shapes, starting with a class *Shape* from which subclasses are derived by inheritance, is shown in Figure 3.1. Arrows indicate the relation 'inherits'.

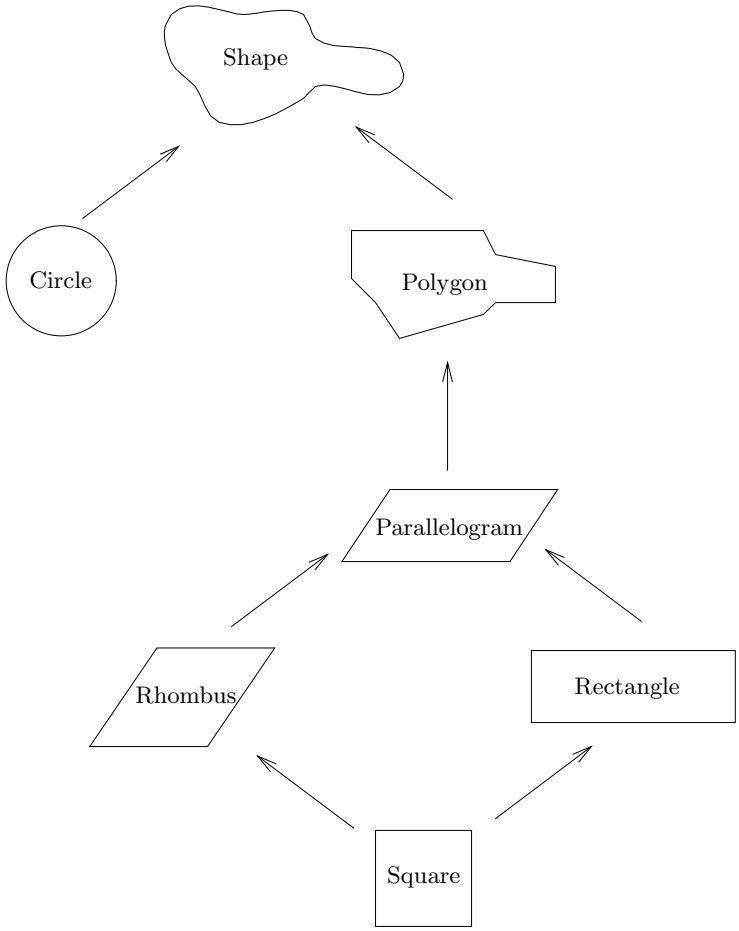


Figure 3.1: The 'shapes' class hierarchy.

In this section, the class *Shape* and the classes derived from, and including, the class *Polygon* will be specified. The class *Circle* will be specified in Section 3.2.

The Shape class

A shape is modelled as having a position in the cartesian plane and a perimeter. The position is given by a vector which is defined in terms of its x- and y-components as follows.

Vector

$x, y : \mathbb{R}$

The unique zero vector with magnitude zero is defined as follows.

$$\mathbf{0} : \mathit{Vector}$$
$$\mathbf{0}.x = \mathbf{0}.y = 0$$

For convenience, the following functions are also defined for vectors.

$|\ \_\ |$ -given a vector v returns the magnitude (length) of v.

rotate -given a vector v and a real number θ returns the vector obtained by rotating v anti-clockwise through angle θ about the origin.

$\_ \perp \_$ -given any two vectors v and w returns true if v is perpendicular to w otherwise returns false.

$\_ + \_$ -given any two vectors v and w returns the vector addition of v and w.

$$|\ \_\ | : \mathit{Vector} \rightarrow \mathbb{R}$$
$$\mathit{rotate} : \mathit{Vector} \times \mathbb{R} \rightarrow \mathit{Vector}$$
$$\_ \perp \_ : \mathit{Vector} \leftrightarrow \mathit{Vector}$$
$$\_ + \_ : \mathit{Vector} \times \mathit{Vector} \rightarrow \mathit{Vector}$$

$$\forall\, v : \mathit{Vector} \bullet$$
$$|\ v\ | = ((v.x)^2 + (v.y)^2)^{1/2}$$
$$\forall\, \theta : \mathbb{R} \bullet$$
$$\mathit{rotate}(v, \theta).x = v.x * \cos(\theta) - v.y * \sin(\theta)$$
$$\mathit{rotate}(v, \theta).y = v.x * \sin(\theta) + v.y * \cos(\theta)$$
$$\forall\, w : \mathit{Vector} \bullet$$
$$v \perp w \Leftrightarrow v.x * w.x + v.y * w.y = 0$$
$$(v + w).x = v.x + w.x$$
$$(v + w).y = v.y + w.y$$

The class *Shape* at the top of the hierarchy of Figure 3.1 is specified as follows.

Shape

$perim : \mathbb{R}$

$position : Vector$

$perim > 0$

Display

$position! : Vector$
$perim! : \mathbb{R}$

$position! = position$
$perim! = perim$

Move

$\Delta(position)$
$v? : Vector$

$position' = position + v?$

Rotate

$\Delta(position)$
$\theta? : \mathbb{R}$

$position' = rotate(position, \theta?)$

The class *Shape* has a constant *perim* denoting the strictly positive length of the perimeter of the shape and a variable *position* denoting the position with respect to the origin of some point of interest of the shape. No initial values of *perim* or *position* are specified.

The class also has three operations. The operation *Display* outputs the values of *position* and *perim* in the output variables *position*! and *perim*! respectively. The operation *Move* corresponds to a translation of the shape by an input vector $v?$ and the operation *Rotate* corresponds to a rotation of the shape about the origin by an input angle $\theta?$. Rotation of a shape about a given centre of rotation $cr : Vector$, not necessarily the origin, can be realised by moving the shape by $-cr$, performing the rotation and then moving the rotated shape by cr.

The Polygon class

A polygon is modelled by its position in the cartesian plane and a finite sequence (*edges*) of at least three straight edges represented by non-zero vectors that sum to zero. Intuitively, the vertices of the polygon are given by the vectors *position*, *position*$+$*edges*(1), *position*$+$*edges*(1)$+$*edges*(2), and so on, as shown in Figure 3.2.

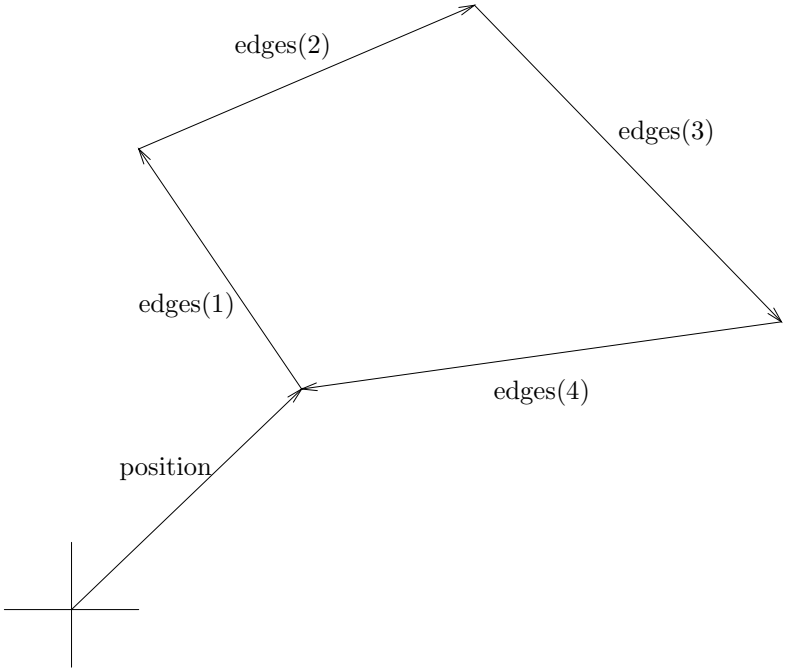


Figure 3.2: Representation of a polygon.

The class *Polygon*, therefore, inherits *Shape* and adds the state variable *edges* and its associated state invariants[2]. The perimeter of a polygon can be determined specifically as the sum of the magnitudes of the edges, and a state invariant is also added to specify this. The operation *Display* is extended to output the sequence of edges of the polygon in the additional output variable *edges*!. The operation *Rotate* is also extended to rotate each edge of the polygon by the input angle θ?.

[2]The symbol $\sum$ is used to represent distributed addition and $\sum$ distributed vector addition.

Polygon

Shape

$$edges : \text{seq } Vector$$

$$\#edges \geqslant 3$$
$$\mathbf{0} \notin \text{ran } edges$$
$$(\sum i : \text{dom } edges \bullet edges(i)) = \mathbf{0}$$
$$perim = \sum i : \text{dom } edges \bullet \mid edges(i) \mid$$

Display

$$edges! : \text{seq } Vector$$

$$edges! = edges$$

Rotate

$$\Delta(edges)$$

$$\#edges' = \#edges$$
$$\forall i : \text{dom } edges \bullet edges'(i) = rotate(edges(i), \theta?)$$

The state schema for *Polygon* is the conjunction of the state schema inherited from *Shape* and the state schema declared explicitly in *Polygon*. A semantically identical schema is shown below.

$$position : Vector$$
$$edges : \text{seq } Vector$$

$$perim > 0$$
$$\#edges \geqslant 3$$
$$\mathbf{0} \notin \text{ran } edges$$
$$(\sum i : \text{dom } edges \bullet edges(i)) = \mathbf{0}$$
$$perim = \sum i : \text{dom } edges \bullet \mid edges(i) \mid$$

The predicate of this schema could be simplified as the predicate $perim > 0$ can be deduced from the other predicates.

The operation *Display* for *Polygon* is defined similarly as the conjunction of the operation *Display* inherited from *Shape* and the operation *Display* declared explicitly in *Polygon*. A semantically identical operation is shown below.

Display

$$
\begin{array}{l}
position! : Vector \\
perim! : \mathbb{R} \\
edges! : \text{seq } Vector \\
\hline
position! = position \\
perim! = perim \\
edges! = edges
\end{array}
$$

Similarly, the operation *Rotate* for *Polygon* is the conjunction of the operation *Rotate* inherited from *Shape* and the operation *Rotate* declared explicitly in *Polygon*. A semantically identical operation is shown below.

Rotate

$$
\begin{array}{l}
\Delta(position, edges) \\
\theta? : \mathbb{R} \\
\hline
position' = rotate(position, \theta?) \\
\#edges' = \#edges \\
\forall\, i : \text{dom } edges \bullet edges'(i) = rotate(edges(i), \theta?)
\end{array}
$$

Redefinition of inherited operations as illustrated by *Display* and *Rotate* may involve the addition of operation parameters and the strengthening of an operation's precondition and postcondition by the addition of constraints. Often, however, it is desirable to remove operation parameters or to weaken an operation's precondition or postcondition. A more general method of redefinition which enables this is presented in Section 3.2.

The Polygon hierarchy

The class *Parallelogram* inherits *Polygon* and adds to the state invariant the condition that the number of edges is four and that the first and third edges sum to zero, i.e. are equal in magnitude but opposite in direction. Since, the condition that the sum of all edges is zero is inherited from *Polygon*, it can be deduced that the second and fourth edges must also sum to zero.

Parallelogram

Polygon

$$
\begin{array}{l}
\#edges = 4 \\
edges(1) + edges(3) = \mathbf{0}
\end{array}
$$

The classes *Rhombus* and *Rectangle* both inherit *Parallelogram*. The class *Rhombus* consists of all shapes of type *Parallelogram* that have the first and second edges equal in magnitude. It can be deduced that all four edges must have equal magnitude.

Rhombus

Parallelogram

$$| \mathit{edges}(1) | = | \mathit{edges}(2) |$$

The class *Rectangle* consists of all shapes of type *Parallelogram* that have the first and second edges perpendicular. It can be deduced that all adjacent edges are perpendicular.

Rectangle

Parallelogram

$$\mathit{edges}(1) \perp \mathit{edges}(2)$$

Multiple Inheritance

The class *Square* illustrates multiple inheritance in Object-Z. A square is a parallelogram which has all edges equal in magnitude and all adjacent edges perpendicular. The class *Square*, therefore, inherits both *Rhombus* and *Rectangle*.

Square

Rhombus
Rectangle

Multiple inheritance, as in the class *Square*, effects the merging of the type, constant and schema definitions in each of the inherited classes with those declared explicitly in the inheriting class. It is identical to progressively inheriting each of the inherited classes in an arbitrary order. The class *Square* could, therefore, be expanded to yield the following semantically identical class. Comparing this class with the previous definition highlights the benefits of reuse obtained through inheritance.

Square

$$perim : \mathbb{R}$$

$$position : Vector$$
$$edges : \text{seq } Vector$$

$$perim > 0$$
$$\#edges \geqslant 3$$
$$\mathbf{0} \notin \text{dom } edges$$
$$(\sum i : \text{dom } edges \bullet edges(i)) = \mathbf{0}$$
$$perim = \sum i : \text{dom } edges \bullet \mid edges(i) \mid$$
$$\#edges = 4$$
$$edges(1) + edges(3) = \mathbf{0}$$
$$\mid edges(1) \mid = \mid edges(2) \mid$$
$$edges(1) \perp edges(2)$$

Display

$$position! : Vector$$
$$perim! : \mathbb{R}$$
$$edges! : \text{seq } Vector$$

$$position! = position$$
$$perim! = perim$$
$$edges! = edges$$

Move

$$\Delta(position)$$
$$v? : Vector$$

$$position' = position + v?$$

Rotate

$$\Delta(position, edges)$$
$$\theta? : \mathbb{R}$$

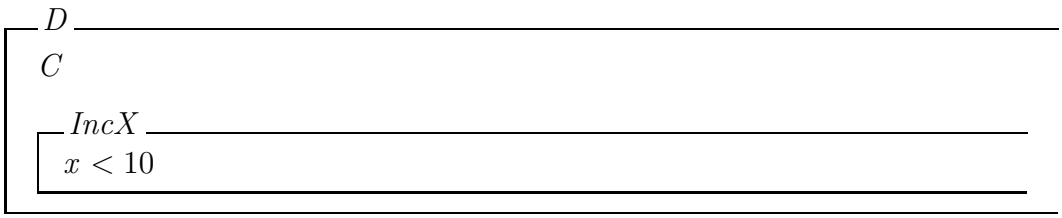
$$position' = rotate(position, \theta?)$$
$$\#edges' = \#edges$$
$$\forall i : \text{dom } edges \bullet edges'(i) = rotate(edges(i), \theta?)$$

The attributes *perim*, *position* and *edges* of the class *Rhombus* are semantically identified with the identically-named attributes in the class *Rectangle* and the state invariants of the inherited classes are conjoined. Also, the operations *Move* and *Rotate* of *Rhombus* are conjoined with the identically-named operations in *Rectangle*.

In general, common-named types or constants occurring in two or more inherited classes are semantically identified and the state schemas, initial state schemas and common-named operations are conjoined. If common-named attributes, operations or operation parameters within common-named operations arise unintentionally, i.e. they are distinct semantically, then renaming in at least one of the classes is required. Renaming of attributes, operations and operation parameters is discussed in Section 3.2.

3.1.3 Inheriting classes with schema expressions

An operation in an inherited class which is defined by a schema expression, or by including other operation schemas, is evaluated, i.e. expanded to a semantically identical schema, before being inherited. For example, consider the class D which inherits the class C of Section 2.1.1 and adds a precondition to the operation $IncX$ stating that the variable x is less than 10.



The operations *IncBoth* and *IncEither*, which are defined by schema expressions involving $IncX$, are unchanged in class D. That is, they are evaluated using the operation $IncX$ of class C and not the extended $IncX$ operation of class D. This convention enables inheritance to be defined semantically as inheriting class C is identical to inheriting a class semantically identical to C but with the operations *IncBoth* and *IncEither* defined in full as operation schemas rather than as schema expressions. The semantics of inheritance, however, is not dealt with in this thesis.

3.2 Renaming and Redefinition

Inheritance, as defined in Section 3.1, enables the addition of attributes, operations, operation parameters and constraints to an inherited class, but does not allow classes to be reused as effectively as may be desired in many applications. In this section, renaming of inherited attributes, operations and operation parameters, and arbitrary redefinition of inherited operations is examined.

Renaming allows name clashes to be avoided when inheriting more than one class and also allows attributes, operations and operation parameters in a subclass to be given more meaningful names. Arbitrary redefinition of operations allows the removal, as well as the addition, of operation parameters and constraints.

3.2.1 Renaming

Renaming can be used to avoid name clashes by enabling the disassociation of common-named, but semantically distinct, attributes and operations in two or more inherited classes. It also enables more meaningful names to be given to inherited attributes and operations when a class is specialised for a particular application.

In Object-Z, renaming is achieved syntactically by a *rename list*. A rename list is a list of substitutions of the form *new name*/*old name* or *operation name*[*new name*/*old name*] for renaming attributes and operations, and operation parameters respectively. The list is in square brackets following the name of the inherited class in which the renaming is to take place. Substitutions within a rename list are separated by commas and may be in any order.

As an example, consider the class *Circle* of the 'shapes' hierarchy of Figure 3.1. A circle can be modelled by the position of its centre in the cartesian plane and its radius as shown in Figure 3.3. The perimeter of a circle is called its circumference and is equal to the product of $2\,\pi$ with its radius.

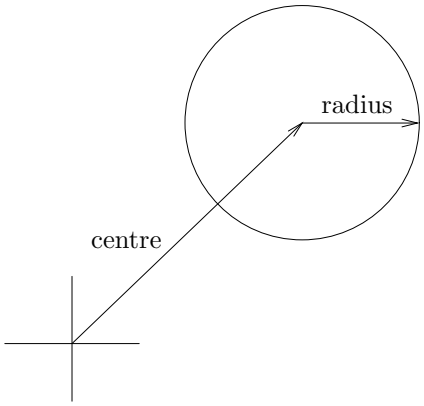
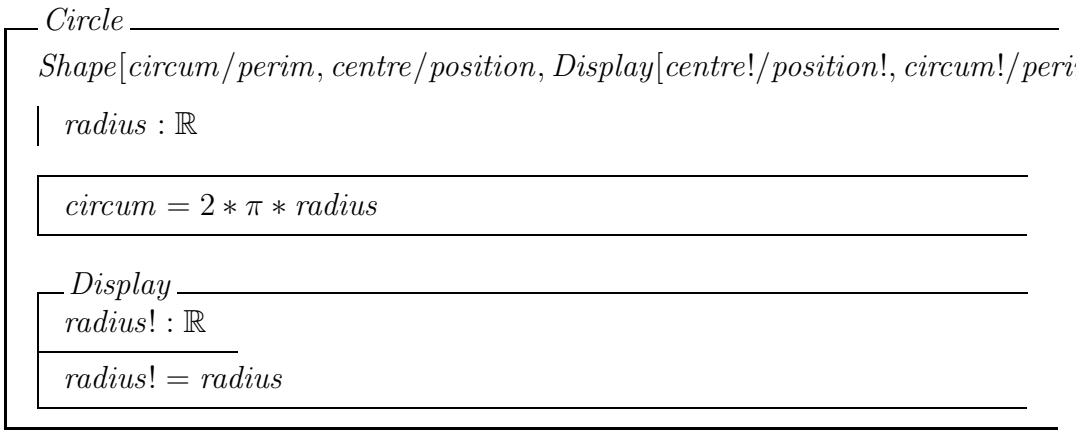


Figure 3.3: Representation of a circle.

The class *Circle* is specified as inheriting the class *Shape* with the state variable *position* renamed to *centre* and the constant *perim* renamed to *circum*. A constant *radius*, representing the radius, is also introduced and related to *circum* by the state invariant. The operation *Display* is extended to output the radius of the circle in the additional output parameter *radius*!. The output parameters *position*! and *perim*! of *Display* are renamed to *centre*! and *circum*! respectively.



Each occurrence of the state variable *position* in *Shape*, either in the state invariant or the Δ-list or predicate of an operation, is replaced with *centre* in class *Circle*. Similarly, each occurrence of the constant *perim* is replaced with *circum*. Also, each occurrence of the parameter *position*! in either the signature or predicate of the operation *Display* in *Shape* is replaced by *centre*! in *Circle* and each occurrence of the parameter *perim*! with *circum*!. The class *Circle* is, therefore, semantically identical to the following class.

Circle

$circum, radius : \mathbb{R}$

$centre : Vector$

$circum > 0$
$circum = 2 * \pi * radius$

Display

$centre! : Vector$
$circum!, radius! : \mathbb{R}$

$centre! = centre$
$circum! = circum$
$radius! = radius$

Move

$\Delta(centre)$
$v? : Vector$

$centre' = centre + v?$

Rotate

$\Delta(centre)$
$\theta? : \mathbb{R}$

$centre' = rotate(centre, \theta?)$

The substitutions in a rename list are not evaluated in any particular order. Each is evaluated with respect to the original inherited class. Therefore, when both an inherited operation and a parameter of that operation are renamed, the old name of the operation is used in the substitution which renames the parameter.

As an example, consider a simple communications protocol which enables the transmission of a sequences of messages. Messages are *accepted* by the protocol and then *delivered* in the same order.

Let *MSG* denote the set of all possible messages that can be transmitted. The protocol can be specified as inheriting the class $Queue[T]$ of Section 2.1.2 with the formal generic parameter T instantiated with *MSG*. The operations *Join* and *Leave* are renamed to *Accept* and *Deliver* respectively, and the operation parameters *item*? and *item*! of *Join* and *Leave* are renamed, respectively, to *msg*? and *msg*!.

Protocol

$Queue[MSG][Accept/Join, Deliver/Leave, Join[msg?/item?], Leave[msg!/item!]]$

3.2.2 Redefinition

Inheritance, as defined in Section 3.1, allows inherited operations to be redefined by conjoining them with common-named operations declared explicitly in the inheriting class. This enables parameters and constraints to be added to an inherited operation but does not enable the removal of parameters and constraints. Adding constraints to an operation's predicate corresponds to strengthening the operation's precondition and postcondition. Often, however, it is desirable to weaken an operation's precondition or postcondition. This is effected by the removal of constraints.

In Object-Z, arbitrary redefinition of an inherited operation is enabled by including the name of the operation in a *redefine list*. A redefine list consists of a list of operation names preceded by the keyword **redef**. The list is in square brackets following the name of the inherited class and the rename list if it exists. The operations in the list are separated by commas and may occur in any order.

As an example, consider the following class *HonoursStudent* based on the Eiffel class *HONOURS_STUDENTS* in Section 1.2.2. This class inherits the class *Student* of Section 3.1.1 and redefines the operation *Enrol* so that a student can enrol in up to ten subjects.

HonoursStudent

$Student[\textbf{redef}\ Enrol]$

Enrol

$\Delta(nb\_subjects)$

$nb\_subjects < 10$

$nb\_subjects' = nb\_subjects + 1$

By including the operation name *Enrol* in the redefine list of the inherited class *Student*, the operation *Enrol* inherited from *Student* has an empty Δ-list and signature and no predicate, i.e. its precondition and postcondition are true. Conjoining this operation with the operation *Enrol* declared in *HonoursStudent* results in an operation semantically identical to the latter. Therefore, an inherited operation which is included in a redefine list can be totally and arbitrarily redefined in the inheriting class.

When both a rename and a redefine list follow the name of an inherited class, the rename list is applied first. Therefore, if an inherited operation is to be renamed and also redefined, the new name of the operation must appear in the redefine list.

3.2.3 Cancellation

Cancellation of attributes and operations allows a class to be a generalisation, rather than a specialisation or extension, of a class which it inherits. Although this provides

greater flexibility when reusing classes, it results in a less intuitive definition of inheritance. Furthermore, inheritance hierarchies involving cancellation can usually be restructured so that cancellation is not required. For example, Wegner and Zdonik[117] mention an example of cancellation where a class of flightless birds can be defined as inheriting the class of birds and cancelling the 'fly' attribute. The inheritance structure they suggest is shown in Figure 3.4.

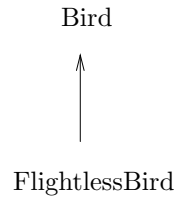


Figure 3.4: 'Bird' hierarchy with cancellation.

An alternative hierarchy is shown in Figure 3.5. The class *Bird* in this hierarchy is equivalent to the class *FlightlessBird* in the hierarchy of Figure 3.4 in that it does not have the attribute 'fly'. The class *FlyingBird* inherits *Bird* and adds the attribute 'fly'.

Figure 3.5: 'Bird' hierarchy without cancellation.

Based on the above discussion, cancellation of attributes and operations is not supported in Object-Z. This is in agreement with most popular object-oriented programming languages which also do not support cancellation.

3.3 Polymorphism

Polymorphism is a mechanism which allows a variable to be declared whose value can be an object from any one of a given collection of classes. Within object-oriented programming languages, it is usual for such a collection of classes to be all those classes that have been derived by inheritance from some common class.

In Object-Z, a variable can be declared, explicitly, to be an object of any class in a particular inheritance hierarchy. For example, if C is a class then the declaration $c : \downarrow C$ declares the object c to be of class C or any class derived from C by inheritance. Adopting the history model of classes, the semantic representation of $\downarrow C$ is the union of all classes derived from, and including, C.

Polymorphism in Object-Z is related to genericity in that it allows a variable to be declared which can be associated with more than one type. Therefore, certain rules have to be obeyed when using a polymorphic variable. Specifically, these rules require that any expression involving the polymorphic variable be applicable to all possible objects to which the variable may be assigned. The responsibility of the specifier to ensure these rules are obeyed can be reduced if inheritance is restricted so that a given class is signature compatible with the classes it inherits.

A class which is signature compatible with a given class can be used in any context in which the given class could be used. That is, any operation invocation allowed for the given class is also allowed for the signature compatible class. Therefore, if an inheritance hierarchy maintains signature compatibility, the specifier has only to ensure that the polymorphic variable is used in a way which an object of the class at the top of the hierarchy could be used.

In Section 3.3.1, polymorphism in Object-Z is illustrated through the specification of a figure consisting of a collection of arbitrary shapes. This example is used to discuss the meaning of references to operations of a polymorphic variable when redefinition occurs in the associated inheritance hierarchy. Rules for maintaining signature compatibility in Object-Z are presented in Section 3.3.2.

3.3.1 Figure example

To illustrate the use of polymorphism in Object-Z consider the following specification of a figure consisting of a collection of arbitrary shapes. Each shape in the figure can be displayed, moved and rotated individually.

Let Id be a set of identifiers used to uniquely identify shapes in the figure. The class *Figure* is specified as follows.

Figure

$$shapes : Id \pfun \downarrow Shape$$

INIT

$$\forall\, id : \mathrm{dom}\, shapes \bullet (shapes, id).INIT$$

$\Phi SelectShape$

$$\Delta(shapes)$$
$$id? : Id$$

$$id? \in \mathrm{dom}\, shapes$$
$$\{id?\} \ndres shapes' = \{id?\} \ndres shapes$$

$$DisplayShape \mathrel{\widehat=} \Phi SelectShape \bullet (shapes, id?).Display$$

$$MoveShape \mathrel{\widehat=} \Phi SelectShape \bullet (shapes, id?).Move$$

$$RotateShape \mathrel{\widehat=} \Phi SelectShape \bullet (shapes, id?).Rotate$$

The class *Figure* has one state variable *shapes* which associates a set of objects of classes in the 'shapes' hierarchy with a unique identifier from the set Id. Initially, each of these objects is in its initial state, i.e. has not undergone any events.

The framing schema $\Phi SelectShape$ specifies that the single shape identified by the identifier $id?$ is updated and all the other shapes in the figure remain unchanged. It is used in the definitions of the operations *DisplayShape*, *MoveShape* and *RotateShape* which display, move and rotate the selected shape respectively.

The interpretation of the notation $(shape, id?).Display$ in the definition of *DisplayShape* depends on the actual class of the selected shape. The *Display* operation which is applied to the selected shape is that from its actual class. This is not necessarily the *Display* operation of *Shape* because the operation in the actual class of the selected shape may have been redefined. For example, if the selected shape is assigned to an object of class *Polygon* then $(shapes, id?).Display$ can be represented semantically as follows.

$(shapes, id?).Display$

$\Delta(shapes)$
$position! : Vector$
$perim! : \mathbb{R}$
$edges! : \text{seq } Vector$
$id? : Id$

$id? \in \text{dom } shapes \cap \text{dom } shapes'$
$shapes(id?).events \in \text{seq } Event$
$front\ shapes'(id?).states = shapes(id?).states$
$shapes'(id?).events = shapes(id?).events^\frown$
$\quad \langle (\text{`}Display\text{'}, \{\text{`}position!\text{'} \mapsto position!, \text{`}perim!\text{'} \mapsto perim!, \text{`}edges!\text{'} \mapsto edges!\})\rangle$

On the other hand, if the selected shape is assigned to an object of class *Circle* then $(shapes, id?).Display$ can be represented semantically as follows.

$(shapes, id?).Display$

$\Delta(shapes)$
$centre! : Vector$
$circum!, radius! : \mathbb{R}$
$id? : Id$

$id? \in \text{dom } shapes \cap \text{dom } shapes'$
$shapes(id?).events \in \text{seq } Event$
$front\ shapes'(id?).states = shapes(id?).states$
$shapes'(id?).events = shapes(id?).events^\frown$
$\quad \langle (\text{`}Display\text{'}, \{\text{`}centre!\text{'} \mapsto centre!, \text{`}circum!\text{'} \mapsto circum!, \text{`}radius!\text{'} \mapsto radius!\})\rangle$

The interpretations of the notations $(shapes, id?).Move$ and $(shapes, id?).Rotate$, in the definitions of the operations *MoveShape* and *RotateShape* respectively, similarly depend on the actual class of the selected shape.

In some object-oriented programming languages, such as Eiffel[80], when an attribute or operation is renamed in an inheritance hierarchy which is being used polymorphically, any reference to the original name is interpreted as a reference to the new name if the actual class of the polymorphic variable is a class in which renaming of the attribute or operation occurs.

This leads to ambiguity, however, when inheriting more than one class belonging to the same inheritance hierarchy. For example, consider the inheritance hierarchy shown in Figure 3.6. (Arrows indicate the relation 'inherits'.)

Assume A has an operation X which is renamed to Y in B and to Z in C. If a variable is declared to be an object of any class in the hierarchy and its actual class is D then a reference to the operation X will be ambiguous. That is, it may refer either to the operation Y inherited from B or the operation Z inherited from C.

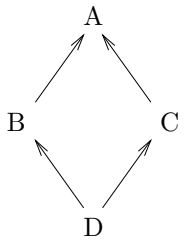


Figure 3.6: Inheritance hierarchy with multiple inheritance.

To avoid such ambiguity in Object-Z, such use of implicit aliases for renamed attributes and operations is not allowed. Instead, aliases may be defined explicitly by the inclusion in a subclass of an additional attribute or operation which is defined identically to the original. In the case of an attribute, the value of the new attribute must always be the same as the original attribute. This can be achieved by a state invariant of the form $a = b$ where a is the original attribute and b its alias.

3.3.2 Signature compatibility

A class is said to be signature compatible with another class if it can be used in any context in which the other class could be used. Therefore, when signature compatibility is maintained in an inheritance hierarchy, a polymorphic variable of that hierarchy can be used in any way that an object of the class at the top of the hierarchy could be used.

Provided renaming does not occur, the rules for maintaining signature compatibility through inheritance are as follows.

Rule 1 - A class must have at least all of the operations of any class it inherits.

Rule 2 - A redefined operation must have exactly the same parameters as the original inherited operation.

Rule 1 is necessary so that any reference made to an operation of a given class is interpretable for its subclasses. This rule is automatically ensured as cancellation of operations is not supported in Object-Z.

Rule 2, however, does not always hold as redefinition of operations enables operation parameters to be added or removed. To illustrate the necessity of Rule 2, consider declaring a variable $s : \downarrow Shape$ and defining the following operations within a class. Both operations are interpretable when the class of the object to which s is assigned is the class *Shape*.

$$
\begin{array}{|l}
\hline
\mathit{Display}_1 \\
\Delta(s) \\
\mathit{position}! : \mathit{Vector} \\
\mathit{perim}! : \mathbb{R} \\
\hline
s.\mathit{Display} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\mathit{Display}_2 \\
s.\mathit{Display} \\
\mathit{out}! : \mathbb{R} \\
\hline
\mathit{out}! = \mathit{perim}!+ \mid \mathit{position}! \mid \\
\hline
\end{array}
$$

The operation $\mathit{Display}_1$ is not interpretable when s is assigned to an object of a class where operation parameters are added to the operation *Display*. For example, if s is assigned to an object of class *Polygon* then the output parameter *edges*!, which occurs in the signature of the operation *Display* in *Polygon* and, hence, in the predicate of $\mathit{Display}_1$, is not declared in the signature of $\mathit{Display}_1$.

On the other hand, the operation $\mathit{Display}_2$ is not interpretable when s is assigned to an object of a class where operation parameters are removed from the operation *Display*. For example, if the output parameter *position*!, which occurs in the predicate of $\mathit{Display}_2$, is not declared in the operation *Display* of the actual class of s then it will also not be declared in the signature of $\mathit{Display}_2$. Similarly for the output parameter *perim*!.

To maintain signature compatibility in an inheritance hierarchy in Object-Z, therefore, a specifier needs to limit the use of redefinition of operations so that there is no addition or cancellation of operation parameters. A formal definition of signature compatibility can be given using the definition of *ClassStruct* from Section 2.2.1. Intuitively, a class is signature compatible with another if it can undergo any event which the other class can undergo (and possibly more).

$$
\begin{array}{|l}
\mathit{sig\_compat} : \mathit{ClassStruct} \leftrightarrow \mathit{ClassStruct} \\
\hline
\forall\, c_1, c_2 : \mathit{ClassStruct} \bullet \\
\quad c_2 \ \underline{\mathit{sig\_compat}}\ c_1 \Leftrightarrow \mathrm{dom}\, c_1.\mathit{trans} \subseteq \mathrm{dom}\, c_2.\mathit{trans}
\end{array}
$$

To maintain the stronger notion of behavioural compatibility in an inheritance hierarchy, the specifier needs also to restrict the way in which an operation's predicate can be redefined. A formal definition of behavioural compatibility and rules for maintaining behavioural compatibility through inheritance in Object-Z are presented in Chapter 6.

Chapter 4

Liveness

> *"The only way to predict the future is to have the power to shape the future."*
>
> — Eric Hoffer
> *The Passionate State of Mind*, 1954.

Classes describe objects in terms of their state and associated operations. The properties they can represent, therefore, are identical to the properties that can be represented by a state transition system. These properties are referred to as safety properties. They specify which state changes may occur but do not require that any state changes actually do occur. Properties which state that a state change, or operation, must occur are referred to as liveness properties.

Liveness properties include guaranteed occurrence of operations, *fairness* and *termination*. Fairness properties state that an operation which is either repeatedly or continuously enabled must eventually occur. Termination properties state that a state from which no further operations can occur, i.e. in which no operations are enabled, must eventually be reached.

Although liveness properties can be expressed in ordinary predicate logic (e.g. see Duke *et al.*[40]), the readability of such properties benefits from the use of temporal logic notation[92, 84, 72]. Temporal logics include, in addition to the propositional operators in ordinary predicate logic, operators corresponding to temporal concepts such as "eventually" and "always".

Object-Z allows the specification of liveness properties by associating with each class a temporal logic history invariant. The history invariant restricts the set of histories derived from the state and operations of the class. A formal definition, in Z, of safety and liveness properties is given in Section 4.1. Section 4.2 presents a temporal logic notation which enables the specification of safety and liveness properties which refer to the occurrence

of both states and operations. Section 4.3 looks at extending Object-Z classes to include temporal logic *history invariants*, i.e. temporal logic predicates which describe an invariant property of the histories of a class.

4.1 Formalising Safety and Liveness

In order to give a clear understanding of the concepts of safety and liveness, formal definitions, in Z, of safety and liveness properties are presented in this section. The definitions are based on definitions given by Alpern and Schneider in [6]. Alpern and Schneider model a property as a set of sequences of states. The understanding is that a property holds for a system when the possible sequences of states through which the system can pass is a subset of the set of sequences of states representing the property.

In this section, an alternative model is presented which allows properties to be concerned with the events a system undergoes, as well as the states through which it passes. Formal definitions of safety and liveness properties are presented for this model in Section 4.1.1. A proof that all properties are the intersection of safety and liveness properties is presented in Section 4.1.2. The definitions and proof correspond precisely to the definitions and proof given for the state sequence model by Alpern and Schneider.

4.1.1 Safety and liveness properties

Adopting the definition of *History* from Section 2.2.2, a property may be defined as a set of histories as follows.

$$Property == \mathbb{P}\, History$$

Intuitively, a property $p : Property$ holds for a system when each possible sequence of states the system can pass through, together with the associated sequence of events the system undergoes, corresponds to a history in the set p.

Safety properties

Intuitively, a safety property states that something 'bad' does not happen. Therefore, if a (possibly infinite) history satisfies a given safety property, nothing 'bad' has happened in the history. Consequently, any pre-history of the history will also satisfy the safety property. Let *prehist* be defined as in Section 2.2.2. The set of safety properties is defined as follows.

$$Safety == \{p : Property \mid \forall\, h : p \bullet prehist(h) \subseteq p\}$$

Liveness properties

A finite history is a history corresponding to a finite sequence of events (and, hence, a finite sequence of states).

$$FiniteHistory == \{f : History \mid f.events \in \operatorname{seq} Event\}$$

A liveness property states that something 'good' must eventually happen. Therefore, any finite history can be extended to satisfy a liveness property. The set of all liveness properties is defined as follows.

$$Liveness == \{p : Property \mid \forall f : FiniteHistory \bullet \exists h : p \bullet f \in prehist(h)\}$$

4.1.2 Other properties

All properties which are not safety or liveness properties may be expressed as the intersection of a safety and a liveness property. The following proof of this statement is based on a similar proof by Alpern and Schneider in [6].

Theorem 4.1

Every property is the intersection of a safety and a liveness property.

$$\forall p : Property \bullet \exists s : Safety;\ l : Liveness \bullet p = s \cap l$$

Proof

Let s be the set containing all pre-histories of histories in p.
That is, $s = \{ph : History \mid \exists h : p \bullet ph \in prehist(h)\}$. (1)
From (1), $\forall h : s \bullet prehist(h) \subseteq s$ (2)
and $p \subseteq s$. (3)
From (2) and the definition of *Safety*, $s \in Safety$. (4)
Let $l = \{h : History \mid h \notin (s \setminus p)\}$. (5)
From (5), $p \subseteq l$. (6)
Let $f \in FiniteHistory$.
From (5), $f \notin (s \setminus p) \Rightarrow f \in l$. (7)
From (1), $f \in (s \setminus p) \Rightarrow \exists h : p \bullet f \in prehist(h)$. (8)
From (6) and (8), $f \in (s \setminus p) \Rightarrow \exists h : l \bullet f \in prehist(h)$. (9)
From (5), (7), (9) and the definition of *Liveness*, $l \in Liveness$. (10)
From (5),
$$\begin{aligned} s \cap l &= s \cap \{h : History \mid h \notin (s \setminus p)\} \\ &= s \cap (\{h : History \mid h \notin s\} \cup p) \\ &= (s \cap \{h : History \mid h \notin s\}) \cup (s \cap p) \\ &= s \cap p. \end{aligned}$$ (11)
From (3) and (11), $p = s \cap l$. □

4.2 Temporal Logic

Temporal logics are extensions to ordinary predicate logic which include operators corresponding to temporal, or time-dependent, concepts. Generally, they are used for writing predicates which refer to sequences of states representing the evolution of a single state over time. Such temporal logics are ideal for specifying liveness properties concerned with the occurrence of states.

In this section, a temporal logic notation is presented which is used for writing predicates which refer to histories corresponding to the sequences of states and corresponding sequences of events that a system can undergo. This enables the specification of properties which are concerned with the occurrence of both states and events. The syntax of the temporal logic is presented in Section 4.2.1 and its semantics in Section 4.2.2.

4.2.1 Introduction to temporal logic

In addition to the conventional logical operators, temporal logic notations include temporal operators such as $\Box$ meaning "always" and $\Diamond$ meaning "eventually". Predicates may be constructed as in ordinary predicate logic or by using the temporal operators as follows. If P is a predicate then $\Box P$ and $\Diamond P$ are also predicates. Intuitively, $\Box P$ describes a system for which P is true now and is always true in the future. Similarly, $\Diamond P$ describes a system for which P is true now or is true at some time in the future.

Temporal operators are particularly useful for specifying the dynamic behaviour of systems. For example, consider describing a system where a variable $i : \mathbb{N}$ at some time equals 1 and at some later time equals 2 in ordinary predicate logic. One solution would be to model the time-dependent variable i as a function of time (modelled as natural numbers) rather than simply as a natural number. The system could then be described as follows.

$$\exists\, t, t' : \mathbb{N} \bullet t < t' \land i(t) = 1 \land i(t') = 2$$

A more elegant solution can be obtained using the temporal operator $\Diamond$ as follows.

$$\Diamond(i = 1 \land \Diamond(i = 2))$$

More complex expressions can also be represented by combining the temporal operators. For example, given a predicate P, $\Diamond\Box P$ states that after some time P is always true. Similarly, $\Box\Diamond P$ states that at all times, P must be true at that time or at some time in the future.

The temporal logic notation presented in this section includes, in addition to the temporal operators $\Box$ and $\Diamond$, notation which enables the expression of properties concerned with

the occurrence of operations. The notation Op **enabled** describes a system for which the operation Op is enabled, i.e. for which the precondition of Op is true. The notation Op **occurs** describes a system which undergoes the operation Op as its next event. For example, the following temporal logic predicate describes a system where after some time, if the operation Op is always enabled then it always eventually occurs.

$$\Diamond\Box(Op \textbf{ enabled}) \Rightarrow \Box\Diamond(Op \textbf{ occurs})$$

This condition on Op is known as *weak fairness*. The condition that an operation which is repeatedly enabled must always eventually occur is known as strong fairness. Strong fairness can also be expressed in temporal logic as follows.

$$\Box\Diamond(Op \textbf{ enabled}) \Rightarrow \Box\Diamond(Op \textbf{ occurs})$$

The notations Op **enabled** and Op **occurs** can also be followed by an optional predicate in ordinary predicate logic. This predicate may be used to restrict the value of the parameters of the operation. For example, the following temporal logic predicate describes a system where the operation Op is always enabled provided its input parameter $x?$ is equal to 1.

$$\Box(Op \textbf{ enabled} \mid x? = 1)$$

Similarly, the following temporal logic predicate describes a system where the operation Op eventually occurs with input parameter $x?$ equal to 1 and output parameter $y!$ equal to 2.

$$\Diamond(Op \textbf{ occurs} \mid x? = 1 \land y! = 2)$$

A complete formal syntax of the temporal logic is presented below[1]. The non-terminals Declaration, Predicate and OpnRef corresponding to declarations, predicates in ordinary predicate logic and references to operations respectively, are defined in Appendix A.

$$\begin{array}{lll}
\mathsf{TLPred} & ::= & \forall\, \mathsf{Declaration}\ [\ |\ \mathsf{TLPred}]\ \bullet\ \mathsf{TLPred} \\
& \mid & \exists\, \mathsf{Declaration}\ [\ |\ \mathsf{TLPred}]\ \bullet\ \mathsf{TLPred} \\
& \mid & \exists_1\, \mathsf{Declaration}\ [\ |\ \mathsf{TLPred}]\ \bullet\ \mathsf{TLPred} \\
& \mid & \mathsf{TLPred1}
\end{array}$$

[1]The syntax is given using an extension to Backus-Naur Form (BNF) defined by Spivey in [108]. Optional phrases are enclosed in slanted square brackets. The binding power of each of the temporal operators is the same as that of the logical operator '$\neg$ '.

TLPred1	::=	Predicate
	|	OpnRef **enabled** [| Predicate]
	|	OpnRef **occurs** [| Predicate]
	|	□TLPred1
	|	◇TLPred1
	|	¬ TLPred1
	|	TLPred1 ∧ TLPred1
	|	TLPred1 ∨ TLPred1
	|	TLPred1 ⇒ TLPred1
	|	TLPred1 ⇔ TLPred1
	|	(TLPred1)

4.2.2 Semantics of temporal logic

The semantics of the temporal logic is given in terms of the semantics of ordinary (i.e non-temporal) predicate logic. Let the type *State* be defined as in Section 2.2.1. The function $\mathcal{M}$ gives the meaning of a predicate p in ordinary predicate logic as a partial function which maps states to boolean values, i.e. $\mathcal{M}(p) \in State \nrightarrow \{true, false\}$. Intuitively, the domain of $\mathcal{M}(p)$ is the set of states in which the predicate p is defined and for any state s in the domain of $\mathcal{M}(p)$, the value of $\mathcal{M}(p)$ s is *true* when p is true in s and *false* when p is false in s.

Let the type *History* be defined as in Section 2.2.2. The function $\mathcal{M}$ can be used to define a function $\mathcal{M}'$ which gives the meaning of a temporal logic predicate tl as a partial function which maps histories to boolean values, i.e. $\mathcal{M}'(tl) \in History \nrightarrow \{true, false\}$.

Semantics of temporal logic without quantification

In this section, it is assumed temporal logic predicates do not involve universal or existential quantification. The semantics of temporal logic predicates involving quantification is given in the following section.

To define the semantics of the temporal logic notation, it is necessary to first define the domains, i.e. the sets of histories, on which the various temporal logic predicates are defined[2]. The domain of the function $\mathcal{M}'$ is defined in terms of the domain of the function $\mathcal{M}$ as follows.

• A predicate p in ordinary predicate logic is defined on a history h if p is defined in the first state of h.

[2]The approach in this thesis differs from most existing semantics of temporal logic (e.g. see [84, 72]) which model a state as a total, rather than partial, function from variables to values. While these approaches are simpler, the use of partial functions provides a more intuitive and easily represented model of objects (as illustrated by the *VendingMachine* example in Chapter 2).

$$\mathrm{dom}\,\mathcal{M}'(p) = \{h : \mathit{History} \mid h.\mathit{states}(1) \in \mathrm{dom}\,\mathcal{M}(p)\}$$

Since all states within a history assign values to the same set of identifiers, if p is defined on h, it follows that p is defined in all states of h.

$$\forall\, h : \mathrm{dom}\,\mathcal{M}'(p) \bullet \forall\, i : \mathrm{dom}\, h.\mathit{states} \bullet h.\mathit{states}(i) \in \mathrm{dom}\,\mathcal{M}(p)$$

Let the type *Event* and the associated functions *op* and *params* be defined as in Section 2.2.1 and let the function $\mathcal{E}$ return the set of all events corresponding to an occurrence of an operation Op, i.e. $\mathcal{E}(Op) \in \mathbb{P}\,\mathit{Event}$.

• A predicate of the form Op **enabled** $\mid p$, where Op is an operation and p is a predicate in ordinary predicate logic, is defined on a history h if the conjunction of the precondition of Op[3] with p is defined in the first state of h when that state is extended with the parameters of any event corresponding to an occurrence of Op.

$$\begin{aligned}&\mathrm{dom}\,\mathcal{M}'(Op\ \textbf{enabled} \mid p) = \\ &\quad \{h : \mathit{History} \mid \forall\, e : \mathcal{E}(Op) \bullet (h.\mathit{states}(1) \oplus \mathit{params}(e)) \in \mathrm{dom}\,\mathcal{M}(\mathrm{pre}\ Op \wedge p)\}\end{aligned}$$

If Op **enabled** $\mid p$ is defined on h, it follows that the conjunction of the precondition of Op with p is defined in every state of h when that state is extended with the parameters of any event corresponding to an occurrence of Op.

$$\begin{aligned}&\forall\, h : \mathrm{dom}\,\mathcal{M}'(Op\ \textbf{enabled} \mid p) \bullet \\ &\quad \forall\, i : \mathrm{dom}\, h.\mathit{states};\ e : \mathcal{E}(Op) \bullet (h.\mathit{states}(i) \oplus \mathit{params}(e)) \in \mathrm{dom}\,\mathcal{M}(\mathrm{pre}\ Op \wedge p)\end{aligned}$$

• A predicate of the form Op **occurs** $\mid p$, where Op is an operation and p is a predicate in ordinary predicate logic, is defined on a history h if p is defined in the first state of h when that state is extended with the parameters of any event corresponding to an occurrence of Op.

$$\begin{aligned}&\mathrm{dom}\,\mathcal{M}'(Op\ \textbf{occurs} \mid p) = \\ &\quad \{h : \mathit{History} \mid \forall\, e : \mathcal{E}(Op) \bullet (h.\mathit{states}(1) \oplus \mathit{params}(e)) \in \mathrm{dom}\,\mathcal{M}(p)\}\end{aligned}$$

• A temporal logic predicate formed by preceding the temporal logic predicate tl with a unary operator is defined on any history on which tl is defined.

$$\mathrm{dom}\,\mathcal{M}'(\Box\, tl) = \mathrm{dom}\,\mathcal{M}'(\Diamond tl) = \mathrm{dom}\,\mathcal{M}'(\neg\ tl) = \mathrm{dom}\,\mathcal{M}'(tl)$$

• Similarly, any combination of the temporal logic predicates tl_1 and tl_2 with a binary operator is defined on any history on which both tl_1 and tl_2 are defined.

[3]The Z notation pre Op is used to denote the precondition of an operation Op.

$$\operatorname{dom}\mathcal{M}'(tl_1 \wedge tl_2) = \operatorname{dom}\mathcal{M}'(tl_1 \vee tl_2) = \\ \operatorname{dom}\mathcal{M}'(tl_1 \Rightarrow tl_2) = \operatorname{dom}\mathcal{M}'(tl_1 \Leftrightarrow tl_2) = \operatorname{dom}\mathcal{M}'(tl_1) \cap \operatorname{dom}\mathcal{M}'(tl_2)$$

Any history whose sequences of states and events are suffixes of those of another history is referred to as a *post-history* of that history. Intuitively, a post-history represents the history of an object after a certain point in time.

$$posthist : History \rightarrow \mathbb{P}\, History$$

$$\forall\, h : History \bullet \\ posthist(h) = \{ph : History \mid \exists\, i : 0\,..\,\#(h.events) \bullet \\ ph.states = squash(1..i \lhd h.states) \wedge \\ ph.events = squash(1..i \lhd h.events)\}$$

From the above definitions, it follows that if a temporal logic predicate tl is defined on a history h then it is also defined on any post-history of h.

$$\forall\, h : \operatorname{dom}\mathcal{M}'(tl) \bullet \forall\, ph : posthist(h) \bullet ph \in \operatorname{dom}\mathcal{M}'(tl)$$

The semantics of the temporal logic is as follows.

(1) A predicate p in ordinary predicate logic is true on a history $h : \operatorname{dom}\mathcal{M}'(p)$ if it is true in the first state of h.

$$\mathcal{M}'(p)\ h \Leftrightarrow \mathcal{M}(p)\ h.states(1)$$

(2) A predicate of the form Op **enabled** $\mid p$, where Op is an operation and p is a predicate in ordinary predicate logic, is true on a history $h : \operatorname{dom}\mathcal{M}'(Op\ \textbf{enabled} \mid p)$ if the conjunction of the precondition of Op with p is true in the first state of h when extended with the parameters of at least one event corresponding to an occurrence of Op.

$$\mathcal{M}'(Op\ \textbf{enabled} \mid p)\ h \Leftrightarrow \exists\, e : \mathcal{E}(Op) \bullet \mathcal{M}(\operatorname{pre} Op \wedge p)\ (h.states(1) \oplus params(e))$$

(3) A predicate of the form Op **occurs** $\mid p$, where Op is an operation and p is a predicate in ordinary predicate logic, is true on a history $h : \operatorname{dom}\mathcal{M}'(Op\ \textbf{occurs} \mid p)$ if the first event of h is an event corresponding to an occurrence of Op and p is true in the first state of h extended with the parameters of the first event in h.

$$\mathcal{M}'(Op\ \textbf{occurs} \mid p)\ h \Leftrightarrow h.events \neq \langle\,\rangle \wedge \\ h.events(1) \in \mathcal{E}(Op) \wedge \mathcal{M}(p)\ (h.states(1) \oplus params(h.events(1)))$$

(4) A temporal logic predicate of the form $\Box tl$, where tl is a temporal logic predicate, is true on a history $h : \operatorname{dom}\mathcal{M}'(tl)$ if tl is true on all post-histories of h.

$$\mathcal{M}'(\Box tl)\ h \Leftrightarrow \forall\, ph : posthist(h) \bullet \mathcal{M}'(tl)\ ph$$

(5) A temporal logic predicate of the form $\Diamond tl$, where tl is a temporal logic predicate, is true on a history $h : \operatorname{dom} \mathcal{M}'(tl)$ if tl is true on at least one post-history of h.

$$\mathcal{M}'(\Diamond tl)\ h \Leftrightarrow \exists\, ph : posthist(h) \bullet \mathcal{M}'(tl)\ ph$$

(6) When conventional logical operators occur in temporal logic expressions, they are interpreted as follows.

If tl is a temporal logic predicate and $h \in \operatorname{dom} \mathcal{M}'(tl)$ then the following is true.

$$\mathcal{M}'(\neg\ tl)\ h \Leftrightarrow \neg\ \mathcal{M}'(tl)\ h$$

Similarly, if tl_1 and tl_2 are temporal logic predicates and $h \in \operatorname{dom} \mathcal{M}'(tl_1) \cap \operatorname{dom} \mathcal{M}'(tl_2)$ then the following are true.

$$\begin{array}{l}
\mathcal{M}'(tl_1 \wedge tl_2)\ h \Leftrightarrow \mathcal{M}'(tl_1)\ h \wedge \mathcal{M}'(tl_2)\ h \\
\mathcal{M}'(tl_1 \vee tl_2)\ h \Leftrightarrow \mathcal{M}'(tl_1)\ h \vee \mathcal{M}'(tl_2)\ h \\
\mathcal{M}'(tl_1 \Rightarrow tl_2)\ h \Leftrightarrow \mathcal{M}'(tl_1)\ h \Rightarrow \mathcal{M}'(tl_2)\ h \\
\mathcal{M}'(tl_1 \Leftrightarrow tl_2)\ h \Leftrightarrow (\mathcal{M}(tl_1)\ h \Leftrightarrow \mathcal{M}'(tl_2)\ h)
\end{array}$$

Semantics of temporal logic with quantification

In this section, the semantics of the previous section is extended to include temporal logic predicates involving universal and existential quantification.

Any quantified variable occurring in a temporal logic predicate is considered constant over the history on which the temporal logic predicate is interpreted. Such variables need not be included in the states of the history. Therefore, to interpret the temporal logic predicate on a history, the states of the history must be extended to include the quantified variables as constants.

The function *extend* extends each state in a history to include a constant assignment of values to a particular set of variables.

$$\begin{array}{|l}
extend : History \times State \to History \\
\hline
\forall\, h_1, h_2 : History;\ s : State \bullet \\
\quad h_2 = extend(h_1, s) \Leftrightarrow \\
\qquad \operatorname{dom} h_2.states = \operatorname{dom} h_1.states \\
\qquad \forall\, i : \operatorname{dom} h_2.states \bullet h_2.states(i) = h_1.states(i) \oplus s \\
\qquad \forall\, i : \operatorname{dom} h_2.events \bullet h_2.events(i) = h_1.events(i)
\end{array}$$

Let $\mathcal{S}$ be a function which returns the set of all states that can be constructed from the variables declared in a declaration d, i.e. $\mathcal{S}(d) \in \mathbb{P}\, State$.

• A temporal logic predicate which involves quantification of a set of variables over the temporal logic predicate tl is defined on a history h if tl is defined on all histories which extend the states of h to include a constant assignment of values to the quantified variables.

$$\begin{aligned}&\operatorname{dom} \mathcal{M}'(\forall\, d \bullet tl) = \operatorname{dom} \mathcal{M}'(\exists\, d \bullet tl) = \operatorname{dom} \mathcal{M}'(\exists_1\, d \bullet tl) = \\ &\quad \{h : History \mid \forall\, s : \mathcal{S}(d) \bullet extend(h, s) \in \operatorname{dom} \mathcal{M}'(tl)\}\end{aligned}$$

The semantics of the previous section can be extended as follows.

(7) When the variables in a declaration d are universally quantified over a temporal logic predicate tl, the resulting predicate is true on a history $h : \operatorname{dom} \mathcal{M}'(\forall\, d \bullet tl)$ if tl is true on all histories which extend the states of h to include a constant assignment of values to the quantified variables. (A predicate $\forall\, d \mid tl_1 \bullet tl_2$, where d is a declaration and tl_1 and tl_2 are temporal logic predicates, is semantically identical to $\forall\, d \bullet tl_1 \Rightarrow tl_2$.)

$$\mathcal{M}'(\forall\, d \mid tl_1 \bullet tl_2)\ h \Leftrightarrow \forall\, s : \mathcal{S}(d) \bullet \mathcal{M}'(tl_1 \Rightarrow tl_2)\ extend(h, s)$$

(8) When the variables in a declaration d are existentially quantified over a temporal logic predicate tl, the resulting predicate is true on a history $h : \operatorname{dom} \mathcal{M}'(\exists\, d \bullet tl)$ if tl is true on at least one history which extends the states of h to include a constant assignment of values to the quantified variables. (A predicate $\exists\, d \mid tl_1 \bullet tl_2$, where d is a declaration and tl_1 and tl_2 are temporal logic predicates, is semantically identical to $\exists\, d \bullet tl_1 \wedge tl_2$.)

$$\mathcal{M}'(\exists\, d \mid tl_1 \bullet tl_2)\ h \Leftrightarrow \exists\, s : \mathcal{S}(d) \bullet \mathcal{M}'(tl_1 \wedge tl_2)\ extend(h, s)$$

Temporal logic predicates involving the unique existence quantifier are interpreted similarly.

$$\mathcal{M}'(\exists_1\, d \mid tl_1 \bullet tl_2)\ h \Leftrightarrow \exists_1\, s : \mathcal{S}(d) \bullet \mathcal{M}'(tl_1 \wedge tl_2)\ extend(h, s)$$

4.3 History Invariants

By incorporating the temporal logic notation of Section 4.2 into Object-Z classes, liveness properties concerned with the occurrence of the states and operations of objects can be specified. Syntactically, this is achieved by including an optional temporal logic history invariant below a horizontal line which separates it from the other features of the class. Conceptually, this separation is similar to the separation of predicates from declarations in ordinary Z schemas since the history invariant constrains the set of histories derived from the state and operations of a class.

The syntactic structure of a class with a temporal logic history invariant is as follows.

ClassName[*generic parameters*]

inherited classes
type definitions
constant definitions
state schema
initial state schema
operations

---

history invariant

When a class inherits another class, the history invariants of the classes are conjoined. A class with no explicit history invariant is assumed to have the default history invariant *true*. Section 4.3.1 looks at incorporating history invariants into Object-Z classes and Section 4.3.2 provides an example of their use through the specification of an alternating bit protocol and the verification of its liveness properties. Section 4.3.3 discusses the issue of realisability of Object-Z classes with history invariants.

4.3.1 Introduction to history invariants

As a preliminary example of the use of history invariants in Object-Z, consider extending the class *VendingMachine* of Section 2.2.1 to ensure that a chocolate is always delivered after a customer has inserted coins to the value of one dollar or more. This can be achieved by inheriting *VendingMachine* and adding a history invariant as follows.

LiveVendingMachine

VendingMachine

$\Box(credit \geqslant 100 \Rightarrow \Diamond(Choc\ \mathbf{occurs}))$

The history invariant of the class *LiveVendingMachine* states that whenever the credit is greater than or equal to one dollar, a chocolate will be delivered at that time or some time in the future. This is not required of the class *VendingMachine* which may stop, i.e. undergo no further operations, after the insertion of coins to the value of one dollar or more.

Despite this difference, the history model does not distinguish between *VendingMachine* and *LiveVendingMachine* as objects of both classes can undergo exactly the same histories. To distinguish between such classes an alternative model based on the *total histories*, i.e. the histories over all time, of a class is adopted.

Let $ClassStruct_0$ refer to the definition of *ClassStruct* in Section 2.2.1. The structural model of a class with a history invariant can be specified as follows. (*Property* is as defined in Section 4.1.1.)

$$
\begin{array}{|l}
\textit{ClassStruct} \\
\hline
\textit{ClassStruct}_0 \\
\textit{hist\_inv} : \textit{Property} \\
\hline
\end{array}
$$

Intuitively, the property representing a history invariant consists of those histories on which the temporal logic formula is defined and true according to the semantics given in Section 4.2.2.

Let the function $\mathcal{H}$ be defined as in Section 2.2.2. The set of total histories of a class can be derived from its structural model using the function $\mathcal{TH}$ defined below.

$$
\begin{array}{|l}
\mathcal{TH} : \textit{ClassStruct} \rightarrow \mathbb{P}\, \textit{History} \\
\hline
\forall\, c : \textit{ClassStruct} \bullet \mathcal{TH}(c) = \mathcal{H}(c) \cap c.\textit{hist\_inv} \\
\hline
\end{array}
$$

The set of total histories of the class are those histories which objects of the class can undergo and which also satisfy the history invariant.

Given any history in the set of total histories of a class it is not necessarily true that any pre-history of that history is also in the set. For example, the history corresponding to the insertion of a one dollar coin and the delivery of a chocolate is a total history of *LiveVendingMachine*, but the pre-history of this history corresponding to the insertion of a one dollar coin only is not. Therefore, if the type of a class is chosen to be the set of total histories, objects of the class cannot necessarily be initialised or have single operations applied to them using the dot notation.

As an alternative, an object can be instantiated from the safety property of its class consisting of all pre-histories of its total histories. This safety property can be defined in terms of the set of total histories as follows.

$$
\begin{array}{|l}
\mathcal{TH}_{safe} : \textit{ClassStruct} \rightarrow \mathbb{P}\, \textit{History} \\
\hline
\forall\, c : \textit{ClassStruct} \bullet \mathcal{TH}_{safe}(c) = \{h : \textit{History} \mid \exists\, h' : \mathcal{TH}(c) \bullet h \in \textit{prehist}(h')\} \\
\hline
\end{array}
$$

To ensure the liveness property of an object, when adopting this approach, it is necessary to ensure that the object *progresses*, i.e. continues to extend its history, until it has undergone a history in the set of total histories of its class. This can be achieved by introducing a history invariant into the class in which the object is instantiated.

This history invariant, by ensuring that the object only extends its history, also captures the intuitive notion that an object within a system is a unique object and, hence, has a unique past history. Without the history invariant, the past history of an object could be changed if the class in which the object is instantiated has an operation which includes the object in its Δ-list but does not constrain the object's value in its post-state.

To define the required history invariant, the notion of the *closure* of a sequence of histories is required. Given a sequence of histories s, where each history in the sequence is a pre-history of each history later in the sequence, the closure of s is the smallest history h satisfying the condition that any history in s is a pre-history of h. The closure of a finite sequence of histories will be identical to the final history in that sequence. The existence and uniqueness of the closure of a given sequence s is assumed without formal proof[4].

$$
\begin{array}{|l}
\mathit{closure} : \mathrm{seq}_\infty \mathit{History} \pfun \mathit{History} \\
\hline
\mathrm{dom}\ \mathit{closure} = \{s : \mathrm{seq}_\infty \mathit{History} \mid \forall i, j : \mathrm{dom}\ s \bullet i \leqslant j \Rightarrow s(i) \in \mathit{prehist}(s(j))\} \\
\forall s : \mathrm{dom}\ \mathit{closure};\ h : \mathit{History} \bullet \\
\quad \mathit{closure}(s) = h \Leftrightarrow \\
\quad\quad \forall i : \mathrm{dom}\ s \bullet s(i) \in \mathit{prehist}(h) \\
\quad\quad \forall j : \mathrm{dom}\ h \bullet \exists i : \mathrm{dom}\ s \bullet \#s(i) \geqslant j
\end{array}
$$

Let A be the structural model of a class A and 'a' denote the identifier corresponding to an object $a : A$. The temporal logic predicate $\vec{a}$ is defined for a class in which a is instantiated. It states that the history of a is continually extended, or after some time remains unchanged, and that the closure of the sequence of successive histories of a is a total history of A. It can be defined formally as follows.

$$
\begin{array}{l}
\mathcal{M}'(\vec{a})\ h \Leftrightarrow \exists s : \mathrm{dom}\ \mathit{closure} \bullet \\
\qquad\qquad \mathrm{dom}\ s = \mathrm{dom}\ h.\mathit{states} \land \\
\qquad\qquad \forall i : \mathrm{dom}\ s \bullet s(i) = h.\mathit{states}(i)(\text{`}a\text{'}) \land \\
\qquad\qquad \mathit{closure}(s) \in \mathcal{TH}(\mathsf{A})
\end{array}
$$

Such a predicate can be used in the specification of a system which includes an object of class *LiveVendingMachine* as follows. (Details of the class not relevant to the instantiation of the object are elided.)

$$
\begin{array}{|l}
\textit{System} \\
\hline
\quad \begin{array}{|l}
\quad \mathit{vm} : \mathit{LiveVendingMachine} \\
\quad \ldots
\end{array} \\
\quad \ldots \\
\hline
\quad \vec{vm} \\
\quad \ldots
\end{array}
$$

To allow the instantiation of aggregates of objects with liveness properties, the temporal logic predicate $(\overrightarrow{f, id})$, where $f : Id \pfun A$ is a function from a set of identifiers Id to a

[4]This definition assumes that *History*, which is defined in terms of *Value*, is a subset of *Value*. Such recursive definitions involving infinite structures can lead to problems with well-definedness. This issue needs to be addressed when developing a full formal semantics for Object-Z.

set of objects of class A and $id : Id$ is an identifier in the domain of f, is also defined as follows. (A denotes the structural model of A and 'f' denotes the identifier corresponding to the function f.)

$$\begin{aligned}\mathcal{M}'((\overrightarrow{f, id}))\ h \Leftrightarrow \exists\, s : \operatorname{dom} closure \bullet \\ \operatorname{dom} s = \operatorname{dom} h.states \wedge \\ \forall\, i : \operatorname{dom} s \bullet (id, s(i)) \in h.states(i)(\text{`}f\text{'})) \wedge \\ closure(s) \in \mathcal{TH}(\mathsf{A})\end{aligned}$$

4.3.2 Alternating bit protocol example

The alternating bit protocol ensures the reliable transmission of an ordered sequence of messages over an unreliable medium. Each message which is accepted for transmission is tagged with a bit (either 1 or 0) and then periodically retransmitted until its tag is returned as acknowledgement of its receipt at the receiving end of the medium.

In this section, the alternating bit protocol is specified as a collection of interacting objects. Each object has a liveness property in the form of fairness conditions placed on particular operations. A proof that the liveness properties of the constituent objects ensure the desired liveness property of the protocol is also presented.

Specification of the alternating bit protocol

The alternating bit protocol can be modelled as consisting of four components: a transmitter, a receiver, a message channel and an acknowledgement channel as shown in Figure 4.1.

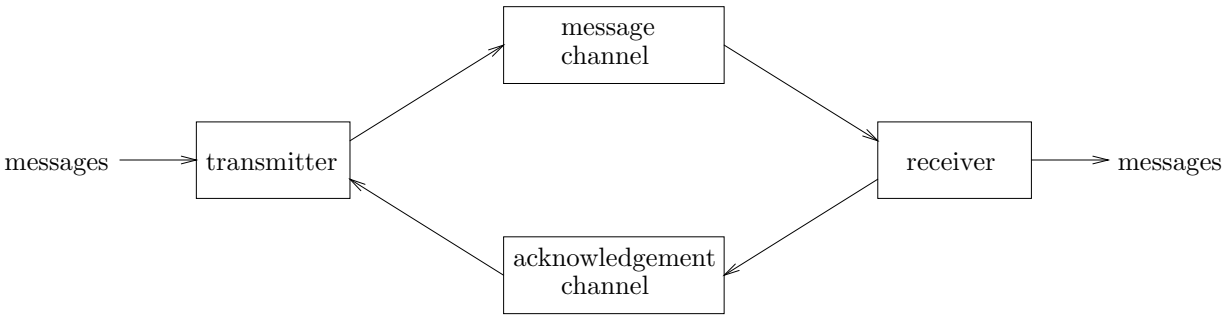


Figure 4.1: Representation of the alternating bit protocol.

The message channel and the acknowledgement channel are unreliable in that they can lose messages and acknowledgements respectively. However, it is assumed they do not lose them indefinitely, i.e. some transmitted messages and acknowledgements must get through.

Let *MSG* denote the set of all messages. Each message is tagged with either a 0 or 1 before transmission. The type *Tag* and *TaggedMSG* are defined as follows.

$$Tag == \{0, 1\}$$
$$TaggedMSG == Tag \times MSG$$

The auxiliary functions *tag* and *msg* are defined to enable access to a tagged message's tag and message components respectively.

$$tag : TaggedMSG \rightarrow Tag$$
$$msg : TaggedMSG \rightarrow MSG$$
$$\forall\, tm : TaggedMSG \bullet tm = (tag(tm), msg(tm))$$

The class *Trans* denoting the transmitter has two state variables: *buf* denoting a buffer which is either empty or contains a tagged message that has been transmitted but not yet acknowledged and *serial* denoting the tag of the last message transmitted. Initially, *buf* is empty and *serial* is set to 0.

Trans

$$buf : \text{seq}\ TaggedMSG$$
$$serial : Tag$$

$$\#buf \leqslant 1$$
$$buf \neq \langle\,\rangle \Rightarrow tag(head\ buf) = serial$$

INIT

$$buf = \langle\,\rangle$$
$$serial = 0$$

TransMsg

$$\Delta(buf, serial)$$
$$msg? : MSG$$
$$tmsg! : TaggedMSG$$

$$buf = \langle\,\rangle$$
$$serial' = 1 - serial$$
$$tmsg! = (msg?, serial')$$
$$buf' = \langle tmsg! \rangle$$

Retrans

$$tmsg! : TaggedMSG$$

$$buf = \langle tmsg! \rangle$$

RecAck

$$\Delta(buf)$$
$$ack? : Tag$$

$$ack? = serial \Rightarrow buf' = \langle\,\rangle$$
$$ack? \neq serial \Rightarrow buf' = buf$$

$$\Diamond\Box(Retrans\ \textbf{enabled}) \Rightarrow \Box\Diamond(Retrans\ \textbf{occurs})$$

The operation *TransMsg* corresponds to the transmitter accepting a message from the environment and transmitting it with an appropriate tag. The tagged message is stored in the transmitter's buffer. The operation *Retrans* corresponds to retransmitting a tagged message in the transmitter's buffer and the operation *RecAck* corresponds to receiving an acknowledgement and emptying the buffer if the acknowledgement is the same as the tag of the last message transmitted.

The history invariant places a condition of weak fairness on the operation *Retrans*. That is, if *Retrans* is continuously enabled it must eventually occur. This ensures the periodic retransmission of any message which has not been acknowledged.

The class *Rec* denoting the receiver has a single state variable *exptag* denoting the tag of the next message the receiver is expecting to deliver. Initially, *exptag* is set to 1, the tag of the first message expected at the receiver.

Rec

$exptag : Tag$

Init

$exptag = 1$

RecMsg

$\Delta(exptag)$
$tmsg? : TaggedMSG$
$msg! : MSG$

$exptag = tag(tmsg?)$
$msg! = msg(tmsg?)$
$exptag' = 1 - exptag$

RejMsg

$tmsg? : TaggedMSG$

$exptag \neq tag(tmsg?)$

TransAck

$ack! : Tag$

$ack! = 1 - exptag$

$\Diamond\Box(TransAck$ **enabled**$) \Rightarrow \Box\Diamond(TransAck$ **occurs**$)$

The operation *RecMsg* corresponds to receiving a message with the expected tag and delivering it to the environment. The operation *RejMsg* corresponds to receiving a message which does not have the expected tag and discarding it. The operation *TransAck* corresponds to transmitting the tag of the last message delivered to the environment.

The history invariant places a condition of weak fairness on the operation *TransAck*. This ensures the periodic retransmission of acknowledgement of the last message received.

As a preliminary to specifying the message and acknowledgement channels, consider extending the class $Queue[T]$ of Section 2.1.2 to specify a lossy queue as follows.

LossyQueue[*T*]

Queue[*T*]

Lose

$\Delta(items)$

$items \neq \langle\,\rangle$

$items' = tail\ items$

$\Box\Diamond(Leave\ \textbf{enabled}) \Rightarrow \Box\Diamond(Leave\ \textbf{occurs})$

The operation *Lose* allows the head of the queue to be lost at any time. This is equivalent to allowing the loss of an element other then the head at some earlier time.

The history invariant places a condition of strong fairness on the operation *Leave*. This ensures that there is no indefinite loss of items from the queue. Strong fairness is required in this case, as opposed to weak fairness, as it is necessary that a *Leave* eventually occurs if it is only repeatedly, and not necessarily continuously, enabled. For example, a history corresponding to an infinite sequence of events where the events correspond to the operations *Join* and *Lose* alternatively never has *Leave* continuously enabled.

The class *MsgChan* denoting the message channel is specified as follows.

MsgChan

LossyQueue[*TaggedMSG*][*Join*[*tmsg*?/*item*?], *Leave*[*tmsg*!/*item*!]]

The class *AckChan* denoting the acknowledgement channel is specified as follows.

AckChan

LossyQueue[*Tag*][*Join*[*ack*?/*item*?], *Leave*[*ack*!/*item*!]]

The class *Protocol* denoting the protocol is specified as follows.

Protocol

$trans : Trans$
$rec : Rec$
$msgchan : MsgChan$
$ackchan : AckChan$

Init

$trans.\textsc{Init}$
$rec.\textsc{Init}$
$msgchan.\textsc{Init}$
$ackchan.\textsc{Init}$

$AcceptMsg \mathrel{\widehat=} trans.TransMsg \parallel msgchan.Join$

$DeliverMsg \mathrel{\widehat=} msgchan.Leave \parallel rec.RecMsg$

$Retrans \mathrel{\widehat=} trans.Retrans \parallel msgchan.Join$

$RejMsg \mathrel{\widehat=} msgchan.Leave \parallel rec.RejMsg$

$TransAck \mathrel{\widehat=} rec.TransAck \parallel ackchan.Join$

$RecAck \mathrel{\widehat=} ackchan.Leave \parallel trans.RecAck$

$LoseMsg \mathrel{\widehat=} msgchan.Lose$

$LoseAck \mathrel{\widehat=} ackchan.Lose$

$\overrightarrow{trans} \wedge \overrightarrow{rec} \wedge \overrightarrow{msgchan} \wedge \overrightarrow{ackchan}$

Objects of the classes *Trans*, *Rec*, *MsgChan* and *AckChan* are effectively instantiated by instantiating objects which satisfy the safety properties of these classes and then ensuring they also satisfy the liveness properties of the classes with a history invariant. Inter-object communication is specified using the $\parallel$ operator.

Verification of liveness

Intuitively, the alternating bit protocol progresses in cycles. Initially, the protocol is ready to accept a message from the environment. If it receives a message, the message is repeatedly retransmitted and, since the message channel does not allow indefinite loss, will be received by the receiver and delivered to the environment. The receiver will then repeatedly transmit an acknowledgement of the receipt of the message and, since the acknowledgement channel does not allow indefinite loss, this acknowledgement will be

received by the transmitter. The protocol is then ready to accept another message from the environment and the cycle repeats.

The completion of a cycle means that the protocol has successfully delivered a message and is ready to accept another message. To deliver a sequence of messages, therefore, the protocol must always complete any cycle it begins. That is, it must be always eventually true that the protocol is ready to accept a new message. A proof that this liveness property holds for the class *Protocol* is given below. The proof assumes the safety properties of the class hold (in particular, that the *buf* attribute of *Trans* is only changed by the operations *TransMsg* and *RecAck*) and that the class has at least one total history.

Theorem 4.2

At any time, the protocol will be ready to accept a message at that time or some time in the future.

$$\Box\Diamond(\mathit{AcceptMsg}\ \textbf{enabled})$$

Proof
Assume the theorem is false. That is, at some time the protocol will not be ready to accept a message at that time or any future time.

That is, $\Diamond\Box\neg$ (*AcceptMsg* **enabled**). (1)
From (1) and the definition of *AcceptMsg*,
$\Diamond\Box\neg$ (*trans.TransMsg* **enabled** $\wedge$ *msgchan.Join* **enabled**). (2)
Since *msgchan.Join* is always enabled, from (2),
$\Diamond\Box\neg$ (*trans.TransMsg* **enabled**). (3)
Consider the object *trans*.
From (3) and the precondition of *TransMsg*, $\Diamond\Box(\mathit{buf} \neq \langle\,\rangle)$. (4)
From (4) and the precondition of *Retrans*, $\Diamond\Box$(*Retrans* **enabled**). (5)
From (5) and the weak fairness condition on *Retrans*, $\Box\Diamond$(*Retrans* **occurs**). (6)
Next consider the object *msgchan*.
From (6), $\Box\Diamond$(*Join* **occurs** | $\mathit{tmsg}? = \mathit{head\ buf}$). (7)
(The expression *head buf*, although strictly in the scope of the object *trans* only, is used throughout the proof to denote a message whose value is equal to *head buf*. Similarly, the expression $\mathit{tag}(\mathit{head\ buf})$ is used to denote an acknowledgement whose value is equal to $\mathit{tag}(\mathit{head\ buf})$.)
From (7) and the definition of *Join*, $\Box\Diamond(\mathit{items}' = \mathit{items} \frown \langle \mathit{head\ buf} \rangle)$. (8)
From the fairness condition on *Leave*, any tagged message at the head of *items* will eventually be removed. Hence, any tagged message in *items* will eventually reach the head of *items*.
Therefore, from (8) and the precondition of *Leave*,
$\Box\Diamond$(*Leave* **enabled** | $\mathit{tmsg}! = \mathit{head\ buf}$). (9)
Furthermore, since after some time *TransMsg* is not enabled (from (3)) any occurrence of *Join* after this time will correspond to *trans* undergoing a *Retrans* operation. Therefore, from the definition of *Retrans*,

$$\Diamond\Box(\mathit{Join}\ \mathbf{occurs} \Rightarrow (\mathit{Join}\ \mathbf{occurs} \mid \mathit{tmsg}? = \mathit{head}\ \mathit{buf})) \quad (10)$$

Therefore, from (10) and the fact that any tagged message in *items* will eventually reach the head of *items*,

$$\Diamond\Box(\mathit{Leave}\ \mathbf{enabled} \Rightarrow (\mathit{Leave}\ \mathbf{enabled} \mid \mathit{tmsg}! = \mathit{head}\ \mathit{buf})). \quad (11)$$

From (9) and the strong fairness condition on *Leave*, it is always true that *Leave* will eventually occur. Since, (11) states that after some time *Leave* is only enabled with $\mathit{tmsg}! = \mathit{head}\ \mathit{buf}$, it is always true that *Leave* will eventually occur with $\mathit{tmsg}! = \mathit{head}\ \mathit{buf}$. That is,

$$\Box\Diamond(\mathit{Leave}\ \mathbf{occurs} \mid \mathit{tmsg}! = \mathit{head}\ \mathit{buf}). \quad (12)$$

Now consider the object *rec*.

From (12), $\Box\Diamond(\mathit{RecMsg}\ \mathbf{occurs} \mid \mathit{tmsg}? = \mathit{headbuf} \vee$
$\mathit{RejMsg}\ \mathbf{occurs} \mid \mathit{tmsg}? = \mathit{headbuf}).$ (13)

From (13) and the definitions of *RecMsg* and *RejMsg*,

$$\Box\Diamond(\mathit{exptag} = 1 - \mathit{tag}(\mathit{head}\ \mathit{buf})). \quad (14)$$

From (13),(14) and the definitions of *RecMsg* and *RejMsg*,

$$\Diamond\Box(\mathit{exptag} = 1 - \mathit{tag}(\mathit{head}\ \mathit{buf})). \quad (15)$$

That is, after *RecMsg* has occurred, $\mathit{exptag} = 1 - \mathit{tag}(\mathit{head}\ \mathit{buf})$ and only *RejMsg*, which does not change the value of *exptag*, is enabled.

From (15) and the precondition of *TransAck*,

$$\Diamond\Box(\mathit{TransAck}\ \mathbf{enabled} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf})). \quad (16)$$

Furthermore, from (15) and the precondition of *TransAck*,

$$\Diamond\Box(\mathit{TransAck}\ \mathbf{enabled} \Rightarrow (\mathit{TransAck}\ \mathbf{enabled} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf}))). \quad (17)$$

From (16) and the weak fairness condition on *TransAck*, it is always true that *TransAck* will eventually occur. Since, (17) states that after some time *TransAck* is only enabled with $\mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf})$, it is always true that *TransAck* will eventually occur with $\mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf})$. That is,

$$\Box\Diamond(\mathit{TransAck}\ \mathbf{occurs} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf})). \quad (18)$$

Now consider the object *ackchan*.

From(18), $\Box\Diamond(\mathit{Join}\ \mathbf{occurs} \mid \mathit{ack}? = \mathit{tag}(\mathit{head}\ \mathit{buf}))$. (19)

From (19) and the definition of *Join*, $\Box\Diamond(\mathit{items}' = \mathit{items} \frown \langle \mathit{tag}(\mathit{head}\ \mathit{buf})\rangle)$. (20)

From the fairness condition on *Leave*, any tagged message at the head of *items* will eventually be removed. Hence, any tagged message in *items* will eventually reach the head of *items*.

Therefore, from (20) and the precondition of *Leave*,

$$\Box\Diamond(\mathit{Leave}\ \mathbf{enabled} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf})). \quad (21)$$

Furthermore, from (17),

$$\Diamond\Box(\mathit{Join}\ \mathbf{occurs} \Rightarrow (\mathit{Join}\ \mathbf{occurs} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf}))). \quad (22)$$

Therefore, from (22) and the fact that any tagged message in *items* will eventually reach the head of *items*,

$$\Diamond\Box(\mathit{Leave}\ \mathbf{enabled} \Rightarrow (\mathit{Leave}\ \mathbf{enabled} \mid \mathit{ack}! = \mathit{tag}(\mathit{head}\ \mathit{buf}))). \quad (23)$$

From (21) and the strong fairness condition on *Leave*, it is always true that *Leave* will eventually occur. Since, (23) states that after some time *Leave* is only enabled with $ack! = tag(head\ buf)$, it is always true that *Leave* will eventually occur with $ack! = tag(head\ buf)$. That is,

$$\Box\Diamond(Leave\ \mathbf{occurs} \mid ack! = tag(head\ buf)). \tag{24}$$

Finally, consider the object *trans* again.
From (24), $\Box\Diamond(RecAck\ \mathbf{occurs} \mid ack? = tag(head\ buf))$. (25)
From (25) and the state invariant of *Trans*,

$$\Box\Diamond(RecAck\ \mathbf{occurs} \mid ack? = serial). \tag{26}$$

From (26) and the definition of *RecAck*, $\Box\Diamond(buf = \langle\,\rangle)$. (27)
Since (27) contradicts (4), the initial assumption is false. □

4.3.3 Realisability

A specification is *realisable*[1] if it does not place constraints on the environment in which it is to operate. An unrealisable specification, which does constrain its environment, is unimplementable.

Allowing the specification of arbitrary liveness properties in Object-Z may lead to specifications which are unrealisable. For example, consider adding the history invariant $\Diamond(Coin\ \mathbf{occurs} \mid coin? = 50)$ to the class *VendingMachine* of Section 2.2.1. The resulting unrealisable specification is unimplementable as it requires the environment, i.e. the user of the vending machine, to eventually insert a 50 cent coin.

Although such specifications cannot be used in the practical development of a system, they are not disallowed in Object-Z as this would decrease the expressibility, and also the simplicity, of the language. It is not uncommon for a specification language to allow specifications to be written which are not implementable. For example, many specification languages, including Z, allow the specification of noncomputable functions.

In general, an object is realisable if it does not constrain, through liveness properties, the occurrence of those operations controlled by its environment. The notion of realisability could, therefore, be formalised in Object-Z if the operations within an object's class were partitioned into those controlled by the environment of the object and those controlled by the object itself. This notion of unilateral control of operations forms the basis of the specification techniques proposed by Abadi and Lamport[1], Lam and Shankar[69] and Lynch and Tuttle[76].

Chapter 5

Full Abstraction

> *"The utmost abstractions are the true weapons with which to control our thought of concrete fact."*
>
> — Alfred North Whitehead
> *Science and the Modern World*, 1925.

A fundamental concept of object orientation is that an object itself may be composed of other objects. The full benefits of such an approach are only realised when the properties of such composite objects can be derived from the properties of their components. This is only possible if the semantic denotation of a class is derivable from the denotations of the classes of the objects of which it is composed. A semantics with this property is described as being *compositional*.

Another desirable property of a semantics of classes is that the denotation of any class is only as detailed as necessary for the semantics to be compositional. A semantics with this property is described as being *fully-abstract*[90, 111, 79]. Intuitively, the denotation of a class in a fully-abstract semantics contains no unnecessary implementational details and, therefore, describes only the *external behaviour* of its objects. Consequently, only objects of classes with equal denotations in such a semantics will be *behaviourally equivalent*, i.e. will behave identically in any context, or environment[1].

Full abstraction has both theoretical and practical significance. Theoretically, a fully-abstract semantics of classes captures the precise meaning of a class independent of its syntactic representation. Practically, a fully-abstract semantics of classes enables simpler definitions of behavioural compatibility, and hence subtyping and refinement, to be developed.

[1] This notion of behavioural equivalence is often referred to as *observational equivalence* in the literature.

Section 5.1 reviews existing approaches to proving that a semantics is fully-abstract and presents an alternative approach for object-oriented languages. This approach can be used to show that a model of classes in Object-Z is fully-abstract. Section 5.2 presents some preliminary models of classes in Object-Z which are not fully-abstract but are used to motivate the definition of a fully-abstract model. Section 5.3 presents the fully-abstract model of classes along with associated proofs of compositionality and full abstraction.

5.1 Introduction to Full Abstraction

A language may be thought of as consisting of components which may be used to construct programs, or systems, in the language, e.g. assignment statements in a sequential programming language or classes in an object-oriented language. A semantics of a language is fully-abstract precisely when any two such components with identical semantic denotations are behaviourally equivalent.

To prove a semantics is fully-abstract, it is necessary to have another semantics of the language from which a definition of behavioural equivalence can be derived. Traditionally, this semantics describes the input/output behaviour of programs, or systems, in the language. Classes in an object-oriented language, however, cannot always be easily described as a relation between input and output. Alternative means of proving a semantics is fully-abstract, therefore, have had to be developed (e.g. see [123]).

Section 5.1.1 examines existing approaches to proving that a semantics is fully-abstract. Section 5.1.2 presents an alternative approach for object-oriented languages and outlines how this approach may be used to show that a model of classes in Object-Z is fully-abstract.

5.1.1 Existing approaches to full abstraction

The most common approach to proving that a semantics of a language is fully-abstract relies on the existence of another semantics of the language which describes the *observable behaviour* of programs, or systems, in the language. The observable behaviour describes only what the system can be observed to do. The external behaviour, as described by a fully-abstract semantics, describes also what the system can refuse to do.

Traditionally, this semantics describes the input/output behaviour of programs, or systems, in the language. Two language components c_1 and c_2 are said to be behaviourally equivalent if the input/output behaviour of the program, or system, formed by placing c_1 in any context C is identical to the input/output behaviour of the program, or system, formed by placing c_2 in C.

To state this more formally, let the notation $C[c]$ represent the program, or system, formed by placing the language component c in the context C. Given a semantics $\mathcal{IO}$

which describes the input/output behaviour of programs, or systems, in a language, two language components c_1 and c_2 are said to be behaviourally equivalent if, for any context C, the following holds.

$$\mathcal{IO}(C[c_1]) = \mathcal{IO}(C[c_2])$$

A semantics $\mathcal{D}$ is said to be fully-abstract with respect to $\mathcal{IO}$ if, for any language components c_1 and c_2 and any context C, the following holds.

$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Leftrightarrow \mathcal{IO}(C[c_1]) = \mathcal{IO}(C[c_2])$$

This approach has been used for many languages. For example, it has been used to define fully-abstract models of nondeterministic dataflow networks in [67] and [97]. It is not, however, applicable for object-oriented languages as it is not always easy to describe a class in terms of a relation between input and output. Classes do not necessarily have any inputs or outputs, in the traditional sense, nor do they necessarily have a final state which could be regarded as an 'output' state. Classes, in fact, fall into the category of *reactive systems* defined by Pnueli in [91]. Such systems are best described in terms of their interaction with their environment.

Yelland[123] proposes a method of proving that a semantics of an object-oriented language is fully-abstract. The approach relies on the existence of another semantics of the language which is compositional but not necessarily fully-abstract. A notion of behavioural equivalence of systems, where systems are collections of interacting objects, is defined in terms of this semantics. This notion is based on the idea that a system can be observed by a "new" object introduced into the system. An observation is made by initialising the variables of the new object with a set of values, executing a sequence of statements and examining the resulting contents of the variables. Two systems are said to be behaviourally equivalent when no observation can distinguish them.

This approach can only be used to develop a semantics which gives fully-abstract denotations to systems and, unlike the approach presented in Section 5.1.2, not necessarily to the classes of the objects of which those systems are composed[2]. Yelland briefly describes an extension to the approach which would allow classes to also be given fully-abstract denotations. This extension relies on having classes identified with systems within the existing semantics.

5.1.2 An alternative approach to full abstraction

This section presents an alternative approach for proving a semantics of an object-oriented language is fully-abstract. The approach is based on the approach presented in Sec-

[2]Yelland's understanding of full abstraction is similar to that of Stoughton[111] which, as pointed out by Meyer in [79], is actually equivalent to compositionality. Stoughton's notion of *contextual full abstraction* is equivalent to the definition of full abstraction adopted in this thesis.

tion 5.1.1 which relies on the existence of another semantics describing observable behaviour.

Adopting the point of view that an object's state is hidden, its observable behaviour consists of the sequence of events it can undergo. Such a sequence of events is referred to as a *trace* of the object. Let $C[c]$ denote the class formed by instantiating an object of the class c in the context, or environment, C. Given a semantics $\mathcal{T}$ which denotes a class by the set of traces its objects can undergo, a semantics $\mathcal{D}$ is fully-abstract with respect to $\mathcal{T}$ if, for any context C, the following holds.

$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Leftrightarrow \mathcal{T}(C[c_1]) = \mathcal{T}(C[c_2])$$

This approach can be used to show that a model of classes in Object-Z is fully-abstract. Adopting the definition of *Event* from Section 2.2.1, a trace can be defined as a possibly infinite sequence of events as follows.

$$Trace == \text{seq}_\infty\ Event$$

Let *ClassStruct* and the function $\mathcal{TH}$ be defined as in Section 4.3.1. The set of traces of a class can be derived from its total histories as defined below.

$$\mathcal{T} : ClassStruct \rightarrow \mathbb{P}\ Trace$$

$$\forall\, c : ClassStruct \bullet \mathcal{T}(c) = \{h : \mathcal{TH}(c) \bullet h.events\}$$

A context in Object-Z can be thought of as an incomplete class schema with all occurrences of a class, used in the declaration of one or more objects, elided. To represent a context, a notation similar to the notation used for specifying Object-Z classes is adopted but with occurrences of the elided class represented by the symbol $\boxdot$. For example, the following is a possible context.

C

$$a, b : \boxdot$$

INIT

$$a.INIT$$
$$b.INIT$$

$$Aop \mathrel{\widehat=} a.Op$$

$$Bop \mathrel{\widehat=} b.Op$$

$$\vec{a} \wedge \vec{b}$$

A class can be placed in a context to form a new class by replacing all occurrences of $\boxdot$ with the class. For example, a class A, which includes an operation Op, can be placed in the above context to yield the class $C[A]$ defined below.

$C[A]$

$a, b : A$

$\textsc{Init}$

$a.\textsc{Init}$
$b.\textsc{Init}$

$Aop \mathrel{\widehat{=}} a.Op$

$Bop \mathrel{\widehat{=}} b.Op$

$\overrightarrow{a} \wedge \overrightarrow{b}$

If a class A can be placed within a particular context C then any class which is signature compatible with A can also be placed in C. If a class B is signature compatible with class A and class A is also signature compatible with class B then A and B are said to be *signature equivalent*. Classes which are signature equivalent can be placed in exactly the same contexts.

Let the relation *sig_compat* be defined as in Section 3.3.2. A definition of signature equivalence can be given in terms of the structural model of classes as follows.

$sig\_equiv : ClassStruct \leftrightarrow ClassStruct$

$$\forall c_1, c_2 : ClassStruct \bullet \\ \quad c_1 \underline{sig\_equiv} c_2 \Leftrightarrow \\ \quad\quad c_1 \underline{sig\_compat} c_2 \\ \quad\quad c_2 \underline{sig\_compat} c_1$$

Given a class A whose structural model is denoted by A, let the structural model of the class $C[A]$, where C is a context, be denoted by $C[\mathsf{A}]$. A model $\mathcal{D}$ of classes is fully-abstract with respect to the trace model if, for all structural models c_1 and c_2 such that $c_1 \underline{sig\_equiv} c_2$, the following holds for all contexts C in which the classes corresponding to c_1 and c_2 can be placed.

$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Leftrightarrow \mathcal{T}(C[c_1]) = \mathcal{T}(C[c_2])$$

5.2 Preliminary Models

Intuitively, the total history model of classes presented in Section 4.3.1 is not fully-abstract with respect to the trace model of Section 5.1.2 as it contains information about the

internal state of objects which cannot be accessed within any context. For example, two classes which are identical except for the name of a particular state variable could not be distinguished by any context but would be given different semantic denotations under the total history model.

The total history model is, however, compositional with respect to the trace model. The ways in which a component object can be referred to in a class are limited by the syntax of Object-Z to $a.\mathit{INIT}$, $\mathrm{pre}\, a.op$, $a.op$ and $\vec{a}$ where a refers to the object and op is an operation. Since each of these constructs are defined only in terms of the total histories of the object's class (as detailed in Sections 2.3.1 and 4.3.1), the set of total histories, and hence the set of traces, of any class can be derived from the sets of total histories of its components.

A fully-abstract model of classes, therefore, can be derived from the total history model by removing exactly that internal state information not required for composition. In this section, two models of classes derived from the total history model are presented as motivation for a fully-abstract model of classes presented in Section 5.3. The first model, presented in Section 5.2.1, is the trace model of Section 5.1.2. It is shown, by means of a counter-example, that this model is not compositional for classes with nondeterministic operations. The second model, presented in Section 5.2.2, attempts to overcome this problem by associating with each trace the operations which are enabled immediately after an object has undergone the sequence of events of the trace. This model is similar to the *readiness* model of Olderog and Hoare[87] and the closely related *failures* model of Brookes *et al.*[22] which has been adopted as the semantics of CSP[61]. It is shown, however, that this model is also not compositional with respect to the trace model for Object-Z classes.

5.2.1 Trace model

A model $\mathcal{D}$ of classes is defined to be compositional if, for all structural models c_1 and c_2 such that c_1 $\underline{sig\_equiv}$ c_2, the following holds for all contexts C in which the classes corresponding to c_1 and c_2 can be placed.

$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Rightarrow \mathcal{D}(C[c_1]) = \mathcal{D}(C[c_2])$$

A model $\mathcal{D}$ of classes is defined to be compositional with respect to the trace model of classes if, for all structural models c_1 and c_2 such that c_1 $\underline{sig\_equiv}$ c_2, the following holds for all contexts C in which the classes corresponding to c_1 and c_2 can be placed.

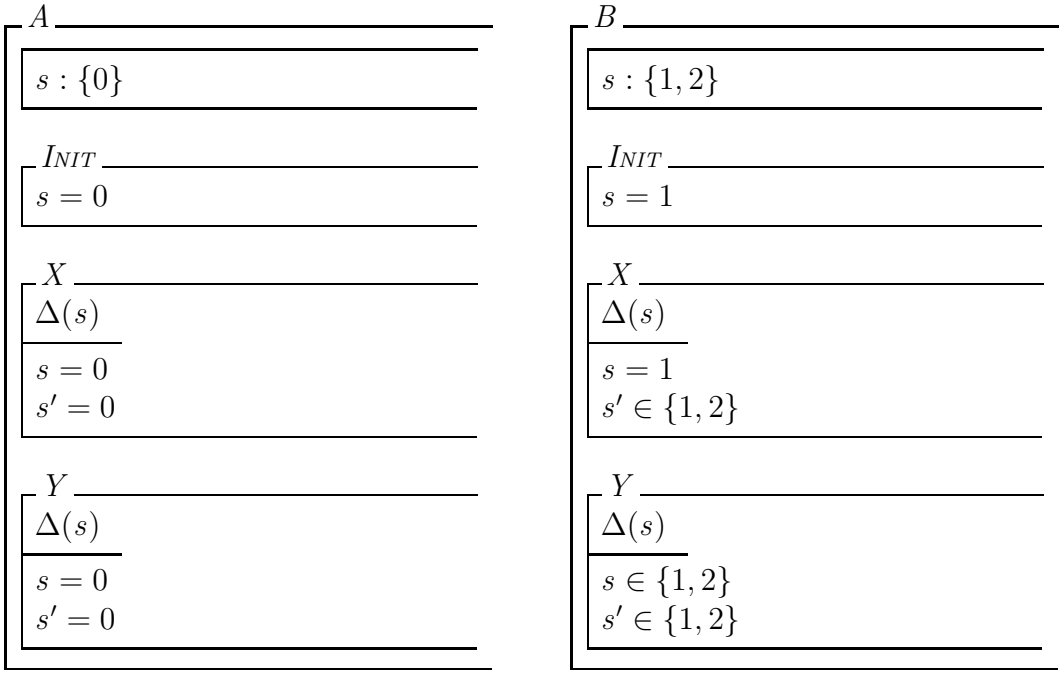
$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Rightarrow \mathcal{T}(C[c_1]) = \mathcal{T}(C[c_2])$$

Equivalently, the following two predicates must hold (see [67]).

$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Rightarrow \mathcal{D}(C[c_1]) = \mathcal{D}(C[c_2])$$
$$\mathcal{D}(c_1) = \mathcal{D}(c_2) \Rightarrow \mathcal{T}(c_1) = \mathcal{T}(c_2)$$

The first predicate states that the model $\mathcal{D}$ is compositional. The second predicate states that the model $\mathcal{D}$ is at least as distinguishing as the trace model $\mathcal{T}$. Therefore, a model of classes which is fully-abstract with respect to the trace model must be at least as distinguishing as the trace model. The trace model itself, therefore, provides the minimum candidate for a fully-abstract model.

The trace model, however, can be shown not to be compositional when nondeterminism is allowed within classes. For example, consider the following signature equivalent Object-Z classes. The operations X and Y in class B are nondeterministic as they have more than one post-state for a given pre-state.



State transition diagrams of these classes are shown in Figure 5.1.

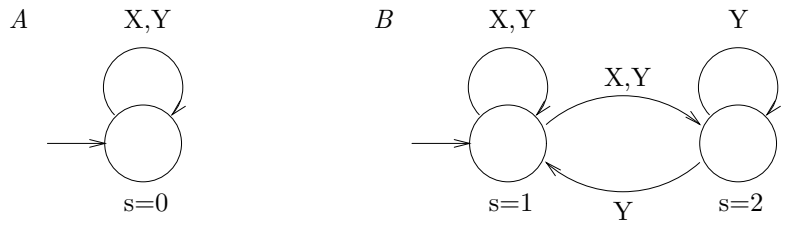


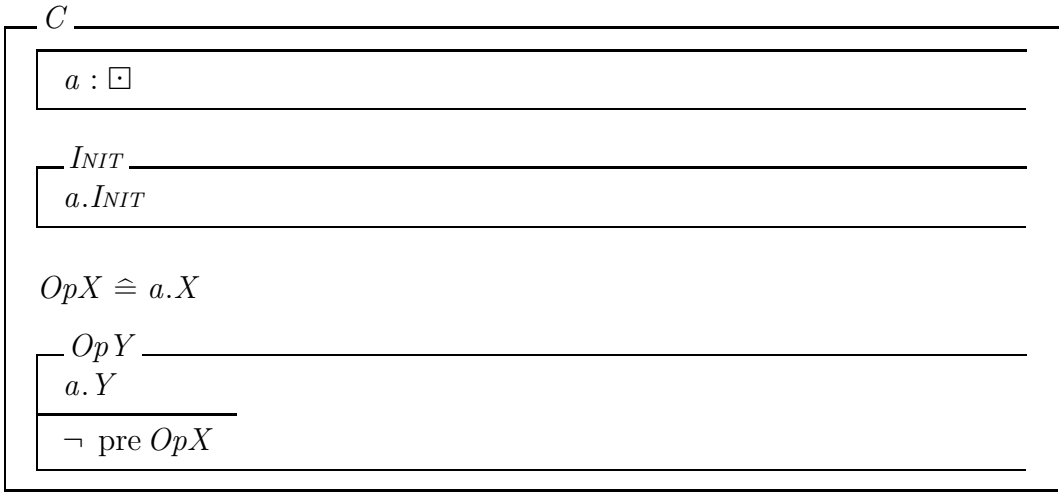
Figure 5.1: State transition diagrams of classes A and B.

The trace models of these classes are identical. That is, each class is represented by the set of all traces made up of events corresponding to the operations X and Y. Let A and

B denote the structural models of classes A and B respectively. The traces of A and B can be defined formally as follows[3].

$$\mathcal{T}(\mathsf{A}) = \mathcal{T}(\mathsf{B}) = \{t : \mathit{Trace} \mid \operatorname{ran} t \subseteq \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\}\}$$

The operation X, however, is always enabled for objects of class A and only sometimes enabled for objects of class B. Therefore, a context which allows, for example, operation Y to occur only when operation X is not enabled and allows operation X to occur otherwise can be used to distinguish classes A and B. This context can be represented as follows.



The traces of the class $C[A]$ consist of events corresponding to the operation OpX only.

$$\mathcal{T}(C[\mathsf{A}]) = \{t : \mathit{Trace} \mid \operatorname{ran} t \subseteq \{(\text{`}OpX\text{'}, \varnothing)\}\}$$

The traces of the class $C[B]$, however, begin with an event corresponding to the operation OpX and then continue with events corresponding to either OpX or OpY.

$$\mathcal{T}(C[\mathsf{B}]) = \{t : \mathit{Trace} \mid t \neq \langle\,\rangle \Rightarrow \mathit{head}\, t = (\text{`}OpX\text{'}, \varnothing) \wedge \operatorname{ran} t \subseteq \{(\text{`}OpX\text{'}, \varnothing), (\text{`}OpY\text{'}, \varnothing)\}\}$$

Since A and B can be distinguished by the context C, the trace model of classes is not compositional.

[3]The events corresponding to the operations X and Y are represented as tuples consisting of the operation's name and an assignment of values to the operation's parameters (as detailed in Section 2.2.1). Since X and Y have no parameters, the associated assignment of values is denoted by the empty set.

5.2.2 Readiness model

The model of classes presented in this section represents an object by the sequence of events it has undergone together with the set of events that it is now ready to perform. The inclusion of this set of events, called the *ready set* of the object, allows the model to distinguish between classes such as A and B of Section 5.2.1.

The model is similar to the readiness model of Olderog and Hoare[87] and the closely related failures model of Brookes *et al.*[22]. The failures model, however, associates with a trace a set of events which can be refused after the trace, rather than the set of events which are enabled after the trace. It is also less distinguishing then the readiness model of Olderog and Hoare and the model presented in this section as it associates with each trace of an object (or process, using the terminology of [22]) not only the set of events it can next refuse, but all subsets of this set. Therefore, while the failures model of an object can be derived from its readiness model, the readiness model of an object cannot necessarily be derived from its failures model.

The failures model and the readiness model of Olderog and Hoare do not include traces corresponding to infinite sequences of events[4] and cannot, therefore, distinguish between systems whose finite behaviour is the same, but whose liveness properties are different. The readiness model presented in this section, however, does include infinite traces.

A *ready-behaviour* is modelled as a (possibly infinite) sequence of events es and a set of events r representing the events which an object is ready to perform after undergoing the events in es. If es is infinite then r is the empty set. Adopting the definition of *Event* from Section 2.2.1, a ready-behaviour can be specified as follows.

$$
\begin{array}{|l}
\textit{ReadyBehaviour} \\
\hline
events : \mathrm{seq}_{\infty}\, Event \\
ready : \mathbb{P}\, Event \\
\hline
events \notin \mathrm{seq}\, Event \Rightarrow ready = \varnothing \\
\hline
\end{array}
$$

The set of events which are ready to occur after an object of a class with structural model c has undergone a history h is given by the function *next* defined below.

[4]The failures model has been extended to include a component of infinite traces by Roscoe and Barrett[95].

$$
\begin{array}{|l}
next : ClassStruct \times History \pfun \mathbb{P}\, Event \\
\hline
\dom next = \{(c, h) : ClassStruct \times History \mid h \in \mathcal{TH}_{safe}(c)\} \\
\forall (c, h) : \dom next \bullet \\
\quad h.events \notin \seq Event \Rightarrow next(c, h) = \varnothing \\
\quad h.events \in \seq Event \Rightarrow \\
\qquad next(c, h) = \{e : Event \mid \exists\, h' : \mathcal{TH}_{safe}(c) \bullet \\
\qquad\qquad front\, h'.states = h.states \land \\
\qquad\qquad h'.events = h.events \frown \langle e \rangle\}
\end{array}
$$

The set of ready-behaviours representing a class can be derived from the total histories of the class using the function $\mathcal{R}$ defined below.

$$
\begin{array}{|l}
\mathcal{R} : ClassStruct \fun \mathbb{P}\, ReadyBehaviour \\
\hline
\forall\, c : ClassStruct \bullet \\
\quad \mathcal{R}(c) = \{r : ReadyBehaviour \mid \exists\, h : \mathcal{TH}(c) \bullet \\
\qquad\qquad r.events = h.events \land \\
\qquad\qquad r.ready = next(c, h)\}
\end{array}
$$

The ready-behaviours of the class A of Section 5.2.1 can be defined formally as follows. (A denotes the structural model of class A.)

$$
\begin{array}{l}
\mathcal{R}(\mathsf{A}) = \{r : ReadyBehaviour \mid \ran r.events \subseteq \{(`X', \varnothing), (`Y', \varnothing)\} \land \\
\qquad\qquad r.events \in \seq Event \Rightarrow \\
\qquad\qquad\quad r.ready = \{(`X', \varnothing), (`Y', \varnothing)\}\}
\end{array}
$$

This is different to the ready-behaviours of class B of Section 5.2.1 which can be defined formally as follows. (B denotes the structural model of class B.)

$$
\begin{array}{l}
\mathcal{R}(\mathsf{B}) = \{r : ReadyBehaviour \mid \ran r.events \subseteq \{(`X', \varnothing), (`Y', \varnothing)\} \land \\
\qquad\qquad r.events \in \seq Event \Rightarrow \\
\qquad\qquad\quad r.ready \in \{\{(`X', \varnothing), (`Y', \varnothing)\}, \{(`Y', \varnothing)\}\} \land \\
\qquad\qquad r.events = \langle\,\rangle \Rightarrow r.ready = \{(`X', \varnothing), (`Y', \varnothing)\}\}
\end{array}
$$

Hence, the readiness model can distinguish between these classes as desired. The readiness model, however, can be shown not to be compositional. For example, consider the following signature equivalent Object-Z classes.

D

$s : \{0, 1, 2\}$

INIT

$s = 0$

X

$\Delta(s)$

$s \in \{0, 2\}$
$s = 0 \Rightarrow s' \in \{0, 1\}$
$s = 2 \Rightarrow s' = 2$

Y

$\Delta(s)$

$s \in \{0, 1, 2\}$
$s = 0 \Rightarrow s' = \{0, 1\}$
$s \in \{1, 2\} \Rightarrow s' = 2$

E

$s : \{3, 4\}$

INIT

$s = 3$

X

$\Delta(s)$

$s = 3$
$s' = \{3, 4\}$

Y

$\Delta(s)$

$s \in \{3, 4\}$
$s' \in \{3, 4\}$

State transition diagrams of these classes are shown in Figure 5.2.

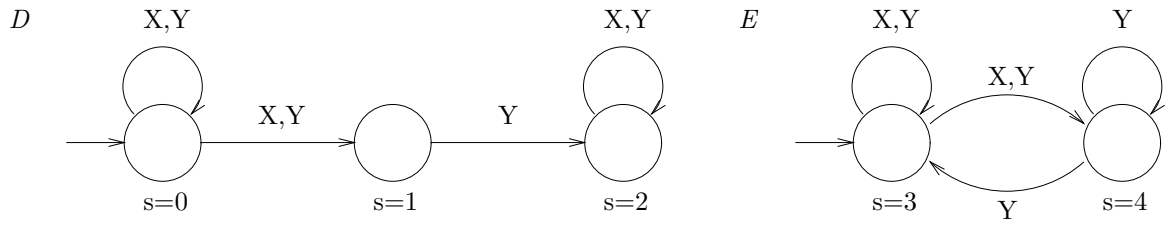


Figure 5.2: State transition diagrams of classes D and E.

The readiness models of these classes are identical. That is, initially an object of either class is ready to perform an X and a Y operation and, after performing any sequence of events, is ready to perform either an X and a Y operation or just a Y operation. Let D and E denote the structural models of classes D and E respectively. The ready-behaviours of D and E can be defined formally as follows.

$$\begin{aligned}\mathcal{R}(\mathsf{D}) = \mathcal{R}(\mathsf{E}) = \{r : \mathit{ReadyBehaviour} \mid \\ \operatorname{ran} r.\mathit{events} \subseteq \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\} \land \\ r.\mathit{events} \in \operatorname{seq} \mathit{Event} \Rightarrow \\ \quad r.\mathit{ready} \in \{\{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\}, \{(\text{`}Y\text{'}, \varnothing)\}\} \land \\ r.\mathit{events} = \langle\,\rangle \Rightarrow r.\mathit{ready} = \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\}\}\end{aligned}$$

The operation X, however, can only be refused at most once for an object of class D but can be refused many times for an object of class E. Therefore, the context C of

Section 5.2.1, which allows operation Y to occur only when operation X is not enabled and allows operation X to occur otherwise, can be used to distinguish classes D and E.

The traces of $C[D]$ begin with an event corresponding to an occurrence of the operation OpX and may have at most one subsequent event corresponding to an occurrence of the operation OpY.

$$\begin{aligned}\mathcal{T}(C[\mathsf{D}]) = \{t : \mathit{Trace} \mid\ & t \neq \langle\,\rangle \Rightarrow \mathit{head}\, t = (\text{`}OpX\text{'}, \varnothing) \land \\ & \operatorname{ran} t \subseteq \{(\text{`}OpX\text{'}, \varnothing), (\text{`}OpY\text{'}, \varnothing)\} \land \\ & \#(t \rhd \{(\text{`}OpY\text{'}, \varnothing)\}) \leqslant 1\}\end{aligned}$$

The traces of $C[E]$, however, begin with an event corresponding to the occurrence of the operation OpX and may have (possibly infinitely) many subsequent events corresponding to occurrences of the operation OpY.

$$\begin{aligned}\mathcal{T}(C[\mathsf{E}]) = \{t : \mathit{Trace} \mid\ & t \neq \langle\,\rangle \Rightarrow \mathit{head}\, t = (\text{`}OpX\text{'}, \varnothing) \land \\ & \operatorname{ran} t \subseteq \{(\text{`}OpX\text{'}, \varnothing), (\text{`}OpY\text{'}, \varnothing)\}\}\end{aligned}$$

Since D and E can be distinguished by the context C, the readiness model of classes is not compositional with respect to the trace model.

5.3 Fully-Abstract Model

A fully-abstract model of classes describes a class in terms of the external behaviour of its objects. It therefore captures the meaning of a class independent of its syntactic representation. In this section, a model of classes in Object-Z is presented which is fully-abstract with respect to the trace model of Section 5.1.2.

The model, called the *complete-readiness* model, represents an object by the sequence of events it has undergone together with the sequence of ready sets at each stage of its evolution. The inclusion of the past ready sets, as well as the current ready set, of the object allows the model to distinguish between classes such as D and E of Section 5.2.2.

The model is presented in Section 5.3.1. It is proved, in Section 5.3.2, to be compositional with respect to the trace model and, in Section 5.3.3, to be fully-abstract with respect to the trace model.

5.3.1 Complete-readiness model

The complete-readiness model represents a class by a set of *behaviours*. A behaviour represents the sequence of events an object has undergone together with the sequence of

ready sets at each stage of its evolution. It can be modelled as a sequence of events *es* and a non-empty sequence of ready sets *rs* such that the number of ready sets in *rs* is one more than the number of events in *es* unless *rs* is infinite in which case *es* is also infinite. Adopting the definition of *Event* from Section 2.2.1, a behaviour can be specified as follows.

$$
\begin{array}{|l}
\textit{Behaviour} \\
\hline
\textit{events} : \mathrm{seq}_\infty \textit{Event} \\
\textit{readys} : \mathrm{seq}_\infty \mathbb{P}\, \textit{Event} \\
\hline
\textit{readys} \neq \langle\,\rangle \\
\forall\, i : \mathbb{N}_1 \bullet i \in \mathrm{dom}\, \textit{events} \Leftrightarrow i + 1 \in \mathrm{dom}\, \textit{readys} \\
\forall\, i : \mathrm{dom}\, \textit{events} \bullet \textit{events}(i) \in \textit{readys}(i) \\
\hline
\end{array}
$$

Let *prehist* be defined as in Section 2.2.2 and the function *next* as in Section 5.2.2. The behaviour of an object of a class with structural model c can be derived from its history using the function $\textit{behav}(c)$ defined below.

$$
\begin{array}{|l}
\textit{behav} : \textit{ClassStruct} \rightarrow (\textit{History} \nrightarrow \textit{Behaviour}) \\
\hline
\forall\, c : \textit{ClassStruct} \bullet \\
\quad \mathrm{dom}\, \textit{behav}(c) = \mathcal{TH}_{\textit{safe}}(c) \\
\quad \forall\, h : \mathrm{dom}\, \textit{behav}(c);\ b : \textit{Behaviour} \bullet \\
\qquad \textit{behav}(c)(h) = b \Leftrightarrow \\
\qquad\quad b.\textit{events} = h.\textit{events} \\
\qquad\quad \forall\, i : \mathrm{dom}\, b.\textit{readys};\ \textit{ph} : \textit{prehist}(h) \bullet \\
\qquad\qquad \#\textit{ph.states} = i \Rightarrow \\
\qquad\qquad\quad b.\textit{readys}(i) = \textit{next}(c, \textit{ph}) \\
\end{array}
$$

The set of behaviours representing a class can be derived from the total histories of the class using the function $\mathcal{CR}$ defined below.

$$
\begin{array}{|l}
\mathcal{CR} : \textit{ClassStruct} \rightarrow \mathbb{P}\, \textit{Behaviour} \\
\hline
\forall\, c : \textit{ClassStruct} \bullet \mathcal{CR}(c) = \textit{behav}(c)(\!|\, \mathcal{TH}(c)\, |\!) \\
\end{array}
$$

The behaviours of the class D of Section 5.2.2 can be defined formally as follows. (D denotes the structural model of class D.)

$$
\begin{aligned}
\mathcal{CR}(\mathsf{D}) = \{ b : \textit{Behaviour} \mid\ & \mathrm{ran}\, b.\textit{events} \subseteq \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\} \wedge \\
& \mathrm{ran}\, b.\textit{readys} \subseteq \{\{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\}, \{(\text{`}Y\text{'}, \varnothing)\}\} \wedge \\
& b.\textit{readys}(1) = \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\} \wedge \\
& \#(b.\textit{readys} \rhd \{(\text{`}Y\text{'}, \varnothing)\}) \leqslant 1\}
\end{aligned}
$$

This is different to the behaviours of class E of Section 5.2.2 which can be defined formally as follows. (E denotes the structural model of class E.)

$$\begin{aligned}\mathcal{CR}(\mathsf{E}) = \{b : Behaviour \mid & \operatorname{ran} b.events \subseteq \{(`X', \varnothing), (`Y', \varnothing)\} \land \\ & \operatorname{ran} b.readys \subseteq \{\{(`X', \varnothing), (`Y', \varnothing)\}, \{(`Y', \varnothing)\}\} \land \\ & b.readys(1) = \{(`X', \varnothing), (`Y', \varnothing)\}\}\end{aligned}$$

That is, an object of class D can refuse to perform an X operation only once whereas an object of class E can refuse to perform an X operation (possibly infinitely) many times. Hence, the complete-readiness model can distinguish these classes as desired.

The proof of full abstraction of the complete-readiness model in the following sections relies on the fact that the precondition of an operation *op* in Object-Z can be tested at any time, i.e. by a statement of the form pre *op*. This ability, inherited from Z, allows the specification of a much wider range of systems than would otherwise be possible.

For example, Object-Z allows an object to be placed in an environment which operates it according to some priority. The context C of Section 5.2.1 which allows operation Y to occur only when operation X is not enabled and allows operation X to occur otherwise is an example of such an environment. Since priority constructs are included in some programming languages, such as occam[66], it is desirable to be able to capture the notion of priority in a specification language. Indeed, alternatives to the failures semantics of CSP have been proposed for this purpose (see e.g. [48]).

The ability to test the precondition of an operation also allows the specification of systems in which an object can perform an operation at any time but must perform it in synchronisation with another object performing an operation whenever that other object can also perform its operation. This notion of composition has been suggested as an alternative to the standard parallel composition operator of process algebras by Pnueli[91]. Pnueli shows that the failures model is not compositional when this type of composition is allowed and suggests a model similar to the complete-readiness model as an alternative.

5.3.2 Proof of compositionality

The complete-readiness model of classes is compositional with respect to the trace model if, for all structural models c_1 and c_2 such that c_1 $\underline{sig\_equiv}$ c_2, the following holds for all contexts C in which the classes corresponding to c_1 and c_2 can be placed.

$$\mathcal{CR}(c_1) = \mathcal{CR}(c_2) \Rightarrow \mathcal{T}(C[c_1]) = \mathcal{T}(C[c_2])$$

In this section, a proof of compositionality is given for any context C which includes a single object a of its elided class as a state variable. The proof could be generalised to also include contexts with multiple objects of the elided class or with aggregates of objects of the elided class. These generalisations are not, however, discussed in this thesis.

The proof relies on the fact that the traces of $C[A]$, for any class A, can be derived from the set of behaviours of A. The ways in which the object a can be referred to within

$C[A]$ are limited by the syntax of Object-Z to $a.\mathit{INIT}$, pre $a.op$, $a.op$ and $\vec{a}$ where op is an operation of class A. The proof, therefore, requires the meanings of these notations to be defined in terms of the complete-readiness model in such a way that the set of traces of $C[A]$ can be derived.

Proof of compositionality for classes without history invariants

In this section, the complete-readiness model is shown to be compositional with respect to the trace model for Object-Z classes without an explicit history invariant, i.e. with an implicit history invariant *true*. The proof is extended in the following section to include classes with explicit history invariants.

Schema definitions of the notations $a.\mathit{INIT}$ and $a.op$ are given in Section 2.3.1. Based on these definitions, the meanings of the notations $a.\mathit{INIT}$, pre $a.op$ and $a.op$ in terms of the total history model are as follows.

- An object a of a class with structural model c satisfies the predicate of $a.\mathit{INIT}$ if and only if its history is in the set $h\_init(c)$ defined below.

$$h\_init(c) = \{h : \mathcal{TH}_{safe}(c) \mid h.events = \langle\,\rangle\}$$

- An object a of a class with structural model c satisfies the predicate pre $a.op$, for a particular assignment of values to the parameters of the operation op, if and only if its history is in the set $h\_pre(c, e)$, where e is the event corresponding to the occurrence of op given the parameter values.

$$\begin{aligned} h\_pre(c, e) = \{h : \mathcal{TH}_{safe}(c) \mid\ & h.events \in \operatorname{seq} Event \land \\ & \exists\, h' : \mathcal{TH}_{safe}(c) \bullet \\ & \quad front\, h'.states = h.states \land \\ & \quad h'.events = h.events \frown \langle e \rangle\} \end{aligned}$$

- The objects a and a' of a class with structural model c satisfy the pre-state and associated post-state of an operation $a.op$, for a particular assignment of values to the parameters of the operation op, if and only if the tuple consisting of the histories of a and a' is in the set $h\_trans(c, e)$, where e is the event corresponding to the occurrence of op given the parameter values.

$$\begin{aligned} h\_trans(c, e) = \{(h, h') : \mathcal{TH}_{safe}(c) \times \mathcal{TH}_{safe}(c) \mid\ & h.events \in \operatorname{seq} Event \land \\ & front\, h'.states = h.states \land \\ & h'.events = h.events \frown \langle e \rangle\} \end{aligned}$$

The meanings of the notations $a.\mathit{INIT}$, pre $a.op$ and $a.op$ can also be defined in terms of the complete-readiness model of classes. Adopting the complete-readiness model, an

object is instantiated from the *safe* behaviours of its class, i.e. the set of all behaviours corresponding to a history in the set of safe total histories of the class.

$$\mathcal{CR}_{safe} : ClassStruct \rightarrow \mathbb{P}\, Behaviour$$

$$\forall\, c : ClassStruct \bullet \mathcal{CR}_{safe}(c) = behav(c)(\!|\, \mathcal{TH}_{safe}(c) \,|\!)$$

The notation $a.\textsc{Init}$, where a is an object, can be represented semantically by the following schema.

$a.\textsc{Init}$

$$a.events = \langle\,\rangle$$

Similarly the notation $a.op$, where a is an object and op an operation in a's class with an input parameter $in? : In$ and an output parameter $out! : Out$, can be represented semantically by the following schema.

$a.op$

$$\Delta(a)$$
$$in? : In$$
$$out! : Out$$

$$a.events \in \operatorname{seq} Event$$
$$front\ a'.readys = a.readys$$
$$a'.events = a.events \frown \langle(\text{`}op\text{'}, \{\text{`}in?\text{'} \mapsto in?, \text{`}out!\text{'} \mapsto out!\})\rangle$$

Therefore, the meanings of the notations $a.\textsc{Init}$, $\operatorname{pre} a.op$ and $a.op$ in terms of the complete-readiness model are as follows.

- An object a of a class with structural model c satisfies the predicate of $a.\textsc{Init}$ if and only if its behaviour is in the set $b\_init(c)$ defined below.

$$b\_init(c) = \{b : \mathcal{CR}_{safe}(c) \mid b.events = \langle\,\rangle\}$$

- An object a of a class with structural model c satisfies the predicate $\operatorname{pre} a.op$, for a particular assignment of values to the parameters of the operation op, if and only if its behaviour is in the set $b\_pre(c, e)$, where e is the event corresponding to the occurrence of op given the parameter values.

$$b\_pre(c, e) = \{b : \mathcal{CR}_{safe}(c) \mid b.events \in \operatorname{seq} Event \wedge \exists\, b' : \mathcal{CR}_{safe}(c) \bullet front\ b'.readys = b.readys \wedge b'.events = b.events \frown \langle e\rangle\}$$

- The objects a and a' of a class with structural model c satisfy the pre-state and associated post-state of an operation $a.op$, for a particular assignment of values to the parameters of the operation op, if and only if the tuple consisting of the behaviours of a and a' is in the set $b\_trans(c, e)$, where e is the event corresponding to the occurrence of op given the parameter values.

$$b\_trans(c, e) = \{(b, b') : \mathcal{CR}_{safe}(c) \times \mathcal{CR}_{safe}(c) \mid b.events \in \operatorname{seq} Event \wedge \\ front\ b'.readys = b.readys \wedge \\ b'.events = b.events \frown \langle e \rangle\}$$

Given a class A with structural model A and a context C which includes a single object a of its elided class as a state variable, consider the following two methods for interpreting constructs in $C[A]$.

Method 1 All component objects of $C[A]$, including a, are instantiated from the set of safe total histories of their classes. All constructs involving these objects are interpreted using the meanings of the constructs in terms of the total history model.

Method 2 The object a is instantiated from the set of safe behaviours of its class. All other component objects of $C[A]$ are instantiated from the set of safe total histories of their classes. All constructs involving a are interpreted using the meanings of the constructs in terms of the complete-readiness model. All constructs involving other component objects are interpreted using the meanings of the constructs in terms of the total history model.

Notice that the set of total histories of $C[A]$ derived using Method 1 will be $\mathcal{TH}(C[\mathsf{A}])$ and, hence, the set of traces of $C[A]$ derived using Method 1 will be $\mathcal{T}(C[\mathsf{A}])$. To show that the complete-readiness model is compositional with respect to the trace model, therefore, it is sufficient to show that the set of traces of $C[A]$ derived using Method 2 is identical to the set of traces of $C[A]$ derived using Method 1. In order to do this, consider the following preliminary definitions which allow a history of $C[A]$ derived using Method 1 to be related to a history of $C[A]$ derived using Method 2.

Given a state s of $C[c]$, where c is the structural model of a class, the state which is identical to s but with the history of a replaced with the behaviour of a can be derived using the function $s\_map(c)$ defined below.

$$s\_map : ClassStruct \rightarrow (State \nrightarrow State)$$

$$\forall\, c : ClassStruct \bullet \\ \quad \operatorname{dom} s\_map(c) = \{s : State \mid \text{`}a\text{'} \in \operatorname{dom} s \wedge s(\text{`}a\text{'}) \in \mathcal{TH}_{safe}(c)\} \\ \quad \forall\, s : \operatorname{dom} s\_map(c) \bullet \\ \quad\quad s\_map(c)(s) = s \oplus \{\text{`}a\text{'} \mapsto behav(c)(s(\text{`}a\text{'}))\}$$

Given a history h of $C[c]$, the history which is identical to h but with the history of a in each state replaced with the behaviour of a can be derived using the function $h\_map(c)$ defined below.

$$
\begin{array}{|l}
h\_map : ClassStruct \rightarrow (History \nrightarrow History) \\
\hline
\forall\, c : ClassStruct \bullet \\
\quad \operatorname{dom} h\_map(c) = \{h : History \mid \operatorname{ran} h.states \subseteq \operatorname{dom} s\_map(c)\} \\
\quad \forall\, h : \operatorname{dom} h\_map(c) \bullet \\
\qquad h\_map(c)(h).events = h.events \\
\qquad \forall\, i : \operatorname{dom} h.states \bullet h\_map(c)(h).states(i) = s\_map(c)(h.states(i))
\end{array}
$$

To prove that the complete-readiness model is compositional with respect to the trace model, it is sufficient to prove that for every history h_1 of $C[A]$ derived using Method 1, the history $h\_map(\mathsf{A})(h_1)$ is a history of $C[A]$ derived using Method 2 and, for every history h_2 of$C[A]$ derived using Method 2, there exists a history in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ which is a history of $C[A]$ derived using Method 1. Since $h\_map(\mathsf{A})$ preserves the trace, i.e the sequence of events, of a history, it follows that the set of traces derived using Method 2 are the same as those derived using Method 1.

A proof of compositionality for classes without explicit history invariants is given below.

Theorem 5.1

Let A be a class and A denote its structural model. Let C be a context which includes a single object a of its elided class as a state variable such that A can be placed in C. Let H_1 be the set of histories of $C[A]$ derived using Method 1, i.e. $H_1 = \mathcal{TH}_{safe}(C[\mathsf{A}])$, and H_2 be the set of histories of $C[A]$ derived using Method 2.

The following predicates are true.

(a) $\forall\, h_1 : H_1 \bullet h\_map(\mathsf{A})(h_1) \in H_2$

(b) $\forall\, h_2 : H_2 \bullet \exists\, h_1 : H_1 \bullet h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$

Proof

(a) The proof is by induction over the length of $h_1.events$.

(i) If $\#h_1.events = 0$ then $h_1.states(1)$ satisfies the predicate of the initial state schema of $C[A]$ using Method 1. Hence, $s\_map(\mathsf{A})(h_1.states(1))$ satisfies the predicate of the initial state schema of $C[A]$ using Method 2 by Lemma C.3[5]. Hence, there exists a history h_2 in H_2 such that $\#h_2.events = 0$ and $h_2.states(1) = s\_map(\mathsf{A})(h_1.states(1))$. That is, $h\_map(\mathsf{A})(h_1) \in H_2$.

[5]The lemmas required for this proof and that of Theorem 5.2 in the next section are included in Appendix C.

(ii) Assume $h\_map(\mathsf{A})(h_1) \in H_2$ for all h_1 such that $\#h_1.events = n$ for some $n \geqslant 0$.

If $\#h_1.events = n + 1$ then the state transition $(h_1.states(n + 1), h_1.states(n + 2))$ is a transition of the event $h_1.events(n + 1)$ using Method 1. Hence, the state transition $(s\_map(\mathsf{A})(h_1.states(n+1)), s\_map(\mathsf{A})(h_1.states(n+2)))$ is a transition of $h_1.events(n+1)$ using Method 2 by Lemma C.6(a).

Since all pre-histories of h_1 are in H_1, there exists a ph_1 in H_1 such that $ph_1 \in prehist(h_1)$ and $\#ph_1.events = n$. Therefore, there exists a ph_2 in H_2 such that $ph_2 = h\_map(\mathsf{A})(ph_1)$ by the above assumption. Therefore, there exists a h_2 (which extends ph_2) in H_2 such that $h_2 = h\_map(\mathsf{A})(h_1)$. Hence, $h\_map(\mathsf{A})(h_1) \in H_2$ for all h_1 such that $\#h_1.events = n + 1$.

(b) The proof is by induction over the length of $h_2.events$.

(i) If $\#h_2.events = 0$ then $h_2.states(1)$ satisfies the predicate of the initial state schema of $C[A]$ using Method 2. Hence, all states in $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(1)\}\ |\!)$ satisfy the predicate of the initial state schema of $C[A]$ using Method 1 by Lemma C.3. Also, since $\mathcal{CR}_{safe}(\mathsf{A}) = behav(\mathsf{A})(\!|\ \mathcal{TH}_{safe}\ |\!)$ and the behaviour of a in $h_2.states(1)$ is in $\mathcal{CR}_{safe}(\mathsf{A})$, $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(1)\}\ |\!)$ is not the empty set. Therefore, there exists a h_1 in H_1 such that $\#h_1.events = 0$ and $h_1.states(1)$ is in $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(1)\}\ |\!)$. That is, there is a h_1 in H_1 such that $h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$.

(ii) Assume there is a h_1 in H_1 such that $h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ for all h_2 such that $\#h_2.events = n$ for some $n \geqslant 0$.

If $\#h_2.events = n + 1$ then the state transition $(h_2.states(n + 1), h_2.states(n + 2))$ is a transition of the event $h_2.events(n + 1)$ using Method 2. Hence, for a given s' in $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(n + 2)\}\ |\!)$, there exists an s in $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(n + 1)\}\ |\!)$ such that (s, s') is a transition of $h_2.events(n+1)$ using Method 1 by Lemma C.6(b). Also, since the behaviour of a in $h_2.states(n+2)$ is in $\mathcal{CR}_{safe}(\mathsf{A})$, $s\_map(\mathsf{A})^{\sim}(\!|\ \{h_2.states(n+2)\}\ |\!)$ is not the empty set.

Since all pre-histories of h_2 are in H_2, there exists a ph_2 in H_2 such that $ph_2 \in prehist(h_2)$ and $\#ph_2.events = n$. Therefore, by the assumption, there exists a ph_1 in H_1 such that $ph_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{ph_2\}\ |\!)$. Hence, by Lemma C.7, there exists a ph_1' in H_1 such that $ph_1'.states(n + 1) = s$ and $h\_map(\mathsf{A})(ph_1') = h\_map(\mathsf{A})(ph_1)$. Therefore, there exists a h_1 (which extends ph_1') in H_1 such that $h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$. Therefore, there is a h_1 in H_1 such that $h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ for all h_2 such that $\#h_2.events = n + 1$. □

Proof of compositionality for classes with history invariants

In this section, the proof of the previous section is extended to include classes with explicit history invariants. Based on the definition presented in Section 4.3.1, the meaning of the notation $\vec{a}$ in terms of the total history model is as follows.

- A history h of $C[c]$, where c is the structural model of a class and C is a context including an object a of its elided class as a state variable, satisfies the history invariant $\vec{a}$ if and only if the sequence of histories of a in h is in the set $h\_seq(c)$ defined below.

$$h\_seq(c) = \{s : \text{seq}_\infty \mathit{History} \mid s \in \text{dom}\, \mathit{closure} \land \mathit{closure}(s) \in \mathcal{TH}(c)\}$$

To define the meaning of $\vec{a}$ in terms of the complete-readiness model, the notion of the closure of a sequence of behaviours is required. The function $b\_closure$ takes as an argument a sequence of behaviours s, where each behaviour in the sequence is a *pre-behaviour* of each behaviour later in the sequence (i.e. its sequences of ready sets and events are are prefixes of those of behaviours later in the sequence). It returns the smallest behaviour b satisfying the condition that any behaviour in s is a pre-behaviour of b. The existence and uniqueness of the closure of a given sequence s is assumed without formal proof.

$$\begin{array}{|l}
b_closure : \text{seq}_\infty \mathit{Behaviour} \pfun \mathit{Behaviour} \\
\hline
\text{dom}\, b_closure = \{s : \text{seq}_\infty \mathit{Behaviour} \mid \forall i, j : \text{dom}\, s \bullet i \leqslant j \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s(i).events \subseteq s(j).events \land \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s(i).readys \subseteq s(j).readys\} \\
\forall s : \text{dom}\, b_closure;\ b : \mathit{Behaviour} \bullet \\
\qquad b_closure(s) = b \Leftrightarrow \\
\qquad\qquad \forall i : \text{dom}\, s \bullet \\
\qquad\qquad\qquad s(i).events \subseteq b.events \\
\qquad\qquad\qquad s(i).readys \subseteq b.readys \\
\qquad\qquad \forall j : \text{dom}\, b \bullet \exists i : \text{dom}\, s \bullet \#s(i) \geqslant j
\end{array}$$

Using this definition, the meaning of the notation $\vec{a}$ in terms of the complete-readiness model is as follows.

- A history h of $C[c]$, where c is the structural model of a class and C is a context including an object a of its elided class as a state variable, satisfies the history invariant $\vec{a}$ if and only if the sequence of behaviours of a in h is in the set $b\_seq(c)$ defined below.

$$b\_seq(c) = \{s : \text{seq}_\infty \mathit{Behaviour} \mid s \in \text{dom}\, b\_closure \land b\_closure(s) \in \mathcal{CR}(c)\}$$

To prove that the complete-readiness model of classes is compositional with respect to the trace model, it is necessary to extend the proof of the previous section to show that for every history h_1 of $C[A]$ which satisfies the history invariant of $C[A]$ using Method 1, the history $h\_map(A)(h_1)$ satisfies the history invariant using Method 2 and for every history h_2 of $C[A]$ which satisfies the history invariant of $C[A]$ using Method 2, there exists a history in $h\_map(A)^{\sim}(\!|\ \{h_2\}\ |\!)$ which satisfies the history invariant using Method 1.

A proof of compositionality for classes with explicit history invariants is given below.

Theorem 5.2

Let A be a class and A denote its structural model. Let C be a context which includes a single object a of its elided class as a state variable such that A can be placed in C. Let H_1' denote the set of histories of $C[A]$ derived using Method 1, i.e. $H_1' = \mathcal{TH}(C[\mathsf{A}])$, and H_2' denote the set of histories of $C[A]$ derived using Method 2.

The following predicates are true.

(a) $\forall\, h_1 : H_1' \bullet h\_map(\mathsf{A})(h_1) \in H_2'$

(b) $\forall\, h_2 : H_2' \bullet \exists\, h_1 : H_1' \bullet h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\, \{h_2\} \,|\!)$

Proof

Let H_1 and H_2 denote the sets of safe histories of $C[A]$ derived using Methods 1 and 2 respectively.

(a) If h_1 is in H_1' then $h_1 \in H_1$ and satisfies the the history invariant of $C[A]$ using Method 1. Therefore, $h\_map(\mathsf{A})(h_1)$ is in H_2 by Theorem 5.1(a) and satisfies the the history invariant of $C[A]$ using Method 2 by Lemma C.10(a). Hence, $h\_map(\mathsf{A})(h_1) \in H_2'$.

(b) If h_2 is in H_2' then $h_2 \in H_2$ and satisfies the history invariant of $C[A]$ using Method 2. Therefore, there exists a history h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\, \{h_2\} \,|\!)$ which is in H_1 and satisfies the history invariant of $C[A]$ using Method 1 by Lemma C.10(b). Therefore, there exists a h_1 in H_1' such that $h_1 \in h\_map(\mathsf{A})^{\sim}(\!|\, \{h_2\} \,|\!)$. □

5.3.3 Proof of full abstraction

Given that the complete-readiness model is compositional with respect to the trace model, it is also fully-abstract with respect to the trace model if, for all structural models c_1 and c_2 such that c_1 $\underline{sig\_equiv}$ c_2, the following holds for all contexts C in which the classes corresponding to c_1 and c_2 can be placed.

$$\mathcal{T}(C[c_1]) = \mathcal{T}(C[c_2]) \Rightarrow \mathcal{CR}(c_1) = \mathcal{CR}(c_2)$$

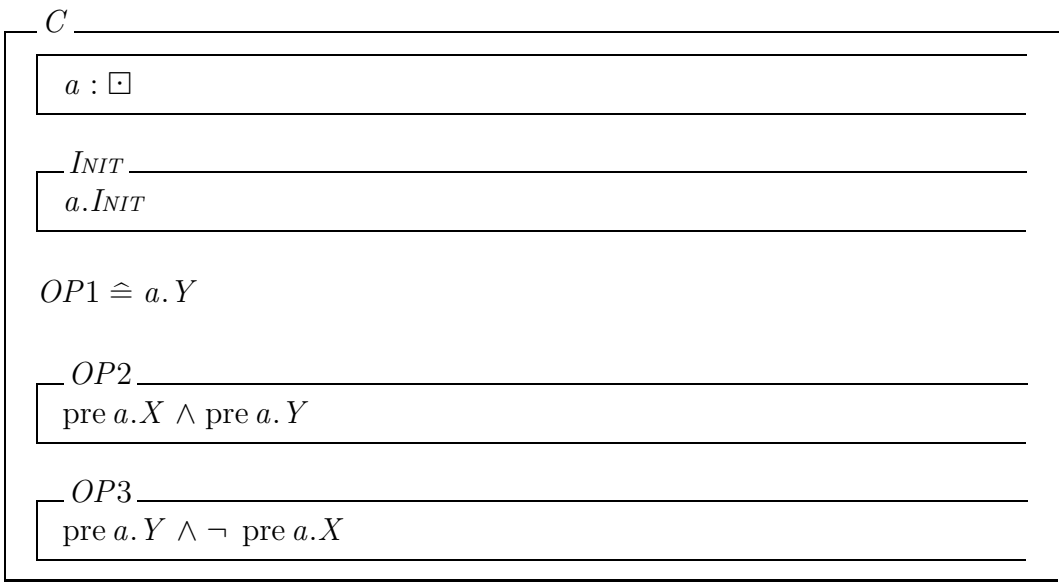
This property states that the complete-readiness model only distinguishes classes when that distinction is necessary for compositionality. Its proof relies on the fact that given any two signature equivalent classes A and B with structural models A and B respectively, if $\mathcal{CR}(\mathsf{A}) \neq \mathcal{CR}(\mathsf{B})$ then a context C can be constructed such that the traces of $C[A]$ are different to those of $C[B]$.

To motivate the following proof, consider again the classes D and E of Section 5.2.2. An object of class E can undergo particular behaviours which an object of class D cannot

undergo. One such behaviour is the behaviour b whose sequences of events and ready sets are as follows.

$$b.events = \langle (\text{`}Y\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing) \rangle$$
$$b.readys = \langle \{(\text{`}X\text{'}, \varnothing), (\text{`}Y\text{'}, \varnothing)\}, \{(\text{`}Y\text{'}, \varnothing)\}, \{(\text{`}Y\text{'}, \varnothing)\} \rangle$$

Consider constructing a context C which has an operation corresponding to each event in $b.events$ and each ready set in $b.readys$ as follows.



The operation $OP1$ corresponds to the occurrence of the events in $b.events$. The operation $OP2$ is enabled when the object a (assumed to be of class D or E) is ready to perform exactly those events in the first ready set of $b.readys$. Similarly, the operation $OP3$ is enabled when a is ready to perform exactly those events in the second and third ready sets of $b.readys$.

The trace $\langle (\text{`}OP2\text{'}, \varnothing), (\text{`}OP1\text{'}, \varnothing), (\text{`}OP3\text{'}, \varnothing), (\text{`}OP1\text{'}, \varnothing), (\text{`}OP3\text{'}, \varnothing) \rangle$ is a possible trace of $C[E]$ since an object of class E can undergo the behaviour b. The trace is not, however, a possible trace of $C[D]$. The context C can, therefore, be used to distinguish classes D and E.

This method can be generalised for any signature equivalent classes with different complete-readiness models as shown in Theorem 5.3 below.

Theorem 5.3

Given two signature equivalent classes A and B with structural models A and B respectively, if $\mathcal{CR}(\mathsf{A}) \neq \mathcal{CR}(\mathsf{B})$ then there exists a context C such that $\mathcal{T}(C[\mathsf{A}]) \neq \mathcal{T}(C[\mathsf{B}])$.

Proof

1) Let C have a single state variable a which is an object of its elided class and which is initialised in its initial state schema.

2) Without loss of generality, assume there is a behaviour b in $\mathcal{CR}(\mathsf{A})$ which is not in $\mathcal{CR}(\mathsf{B})$.

3) For each event e in the range of $b.events$, let C have an operation which corresponds to a undergoing that event.

4) For each ready set r in the range of $b.readys$, let C have an operation which has a true postcondition and a precondition that states that each event in r is enabled and each event of A not in r is not enabled.

Let t be a trace such that

- if $b.events$ is finite then t is finite and $\#t = \#b.readys + \#b.events$, otherwise t is infinite, and
- for all $i : \operatorname{dom} t$, if i is odd then $t(i)$ is the event corresponding to the operation associated with $b.readys((i+1)/2)$ (as described in step 4 above) and if i is even then $t(i)$ is the event corresponding to the operation associated with $b.events(i/2)$ (as described in step 3 above).

The trace t will be in $\mathcal{T}(C[\mathsf{A}])$ but will not be in $\mathcal{T}(C[\mathsf{B}])$. □

Chapter 6

Behavioural Compatibility

> *"A stander-by may sometimes, perhaps, see more of the game than he that plays it."*
>
> — Jonathan Swift
> *A Tritical Essay Upon the Faculties of the Mind*, 1707.

The final step in the formal development of a software system is the refinement of its specification towards an executable implementation. To take advantage of the modular structure of an object-oriented specification, it is desirable to refine the specification by separately refining the classes of each of its component objects. This will result in a valid refinement of the overall specification if each class which refines another class is also a subtype of the other class.

Subtyping is related to object substitutability. A class is a subtype of another class if objects of that class can be substituted for objects of the other class in any system so that the system, after the substitution has occurred, can only behave in ways that it could have behaved before the substitution. The external behaviour of a subtype of a given class is a specialisation of the external behaviour of that class. Subtyping is, therefore, also referred to as *behavioural compatibility* (e.g. see [117]).

The definition of behavioural compatibility for a particular object-oriented language depends on the method of interaction of an object and its environment. In particular, it depends on whether an object is regarded as an *active* entity which undergoes operations autonomously or a *passive* entity which undergoes operations only when directed to do so by its environment. In the former situation, the environment can be thought of as an *observer* and a notion of *observational compatibility* is required. In the latter situation, the environment can be thought of as an *operator* and a notion of *operational compatibility* is required.

Object-Z allows the specification of both active and passive objects. The notion of behavioural compatibility must, therefore, be strong enough to allow for both situations.

Weaker notions of behavioural compatibility are relevant, however, when the environments in which an object can be placed are limited to reflect a particular programming paradigm. Section 6.1 reviews existing definitions of behavioural compatibility in object-oriented languages and shows that such definitions cannot always be used to find all classes which are behaviourally compatible with a given class. An alternative approach to behavioural compatibility is outlined and a definition of behavioural compatibility in Object-Z, based on this approach, is presented. Section 6.2 examines notions of observational and operational compatibility in Object-Z and Section 6.3 looks at rules for maintaining behavioural compatibility through inheritance.

6.1 Introduction to Behavioural Compatibility

Behavioural compatibility is most often referred to in the literature as subtyping. The use of subtyping as a design methodology for modelling conceptual hierarchies in object-oriented programming languages is discussed by Halbert and O'Brien in [54]. Subtyping can also form the basis of a theory of class refinement in object-oriented specification languages.

Traditionally, subtyping has been strongly linked with inheritance in many object-oriented programming languages. In more recent languages and theories of object orientation, however, subtyping and inheritance have been regarded as orthogonal issues (e.g. see [7, 30, 105]). The motivation for this point of view is that inheritance is concerned with the sharing of *internal* structure between classes whereas subtyping is concerned with the specialisation of *external* behaviour.

Section 6.1.1 examines existing definitions of behavioural compatibility and shows that such definitions are, in some cases, stronger than necessary. Section 6.1.2 outlines an alternative approach to behavioural compatibility and presents a definition of behavioural compatibility in Object-Z based on this approach.

6.1.1 Existing approaches to behavioural compatibility

Many early object-oriented programming languages associated subtyping, or behavioural compatibility, with inheritance under the assumption that the sharing of internal structure would lead to the specialisation of external behaviour. However, this assumption is, in general, too restrictive. It is possible for classes with identical behaviour to have unrelated internal structure. For example, a class which behaves like a queue may be modelled internally as an array or, alternatively, as a linked list. Furthermore, if redefinition of operations is allowed through inheritance then a class may behave very differently to the classes it inherits. Definitions of behavioural compatibility which are independent of inheritance have, therefore, been developed.

Most of these definitions of behavioural compatibility are based on relationships between the individual operations of the related classes. According to these definitions, a class B is behaviourally compatible with a class A if it has at least all of the operations of A and there exists a function ϕ between the states of A and the states of B such that for each operation *op* in A the following hold[1].

- If a state s is a pre-state of *op* in A then $\phi(s)$ is a pre-state of *op* in B.
- For a given pre-state s of *op* in A, if a state t' is a post-state of *op* in B when applied in state $\phi(s)$ then all s' such that $\phi(s') = t'$ are post-states of *op* in A when applied in state s.

The above relation on operations, referred to as a *contravariance* relation, ensures that an object of class B can perform a particular operation of class A whenever an object of class A could have performed it. Furthermore, the result of performing the operation at any such instant is a possible result of an object of class A performing the same operation at that instant. The approach is, therefore, applicable for languages where objects are regarded as passive entities which are operated by their environment.

The contravariance approach to behavioural compatibility is widely accepted. Formal definitions of subtyping by America[8] and Utting and Robinson[114] are based on this approach, as is the work on the refinement of objects in the object-oriented extension to Z proposed by Whysall and McDermid[119]. Inheritance in the object-oriented programming language Eiffel[80] is also restricted to maintain behavioural compatibility by adopting a contravariant redefintion rule on the pre-conditions and post-conditions of operations.

The approach, being based on a relationship between individual operations, is not, however, *complete*. That is, not all classes which are behaviourally compatible with a given class can be found using such definitions. For example, consider the following Object-Z classes.

[1]The definition assumes the pre-state of an operation includes its input parameters and the post-state of an operation includes its output parameters.

F

$s : \{0, 1, 2\}$

INIT

$s = 0$

X

$\Delta(s)$

$s \in \{0, 2\}$
$s = 0 \Rightarrow s' = 1$
$s = 2 \Rightarrow s' = 0$

Y

$\Delta(s)$

$s \in \{1, 2\}$
$s = 1 \Rightarrow s' = 0$
$s = 2 \Rightarrow s' = 1$

G

$s : \{0, 1\}$

INIT

$s = 0$

X

$\Delta(s)$

$s = 0$
$s' = 1$

Y

$\Delta(s)$

$s = 1$
$s' = 0$

State transition diagrams of these classes are shown in Figure 6.1.

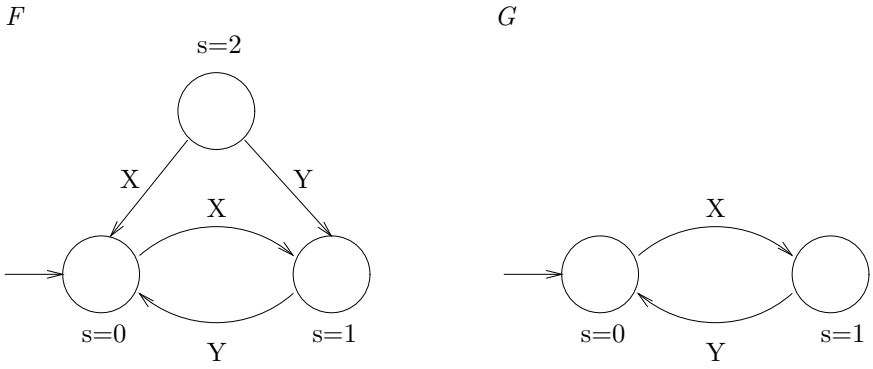


Figure 6.1: State transition diagrams of classes F and G.

An object of class G can perform a particular operation in F whenever an object of class F could have performed it. Furthermore, the result of performing the operation at any such instant is identical to an object of class F performing the same operation at that instant. Therefore, the classes satisfy the notion of behavioural compatibility captured by the contravariance approach. The operations in F are not, however, related by the contravariance relation to the operations in G.

The discrepancy arises because the state $s = 2$, which is a pre-state of both operations in F, is not reachable from the initial state of F. That is, the state $s = 2$ never occurs in any

history of the class F and, therefore, transitions beginning in this state do not need to be considered when determining whether another class is behaviourally compatible with F.

Although classes such as F which have unreachable states would not arise intentionally, they may arise unintentionally through the use of inheritance. For example, the postcondition of an operation could be strengthened during inheritance removing access to a state which previously only occurred as a post-state of that operation. A complete definition of behavioural compatibility, which identifies all classes which are behaviourally compatible with a given class, needs to account for the possibility of unreachable states and cannot, therefore, be based on an approach which examines the operations of a class individually.

6.1.2 An alternative approach to behavioural compatibility

This section presents an alternative approach to behavioural compatibility which can be shown to be complete. That is, the approach can be used to find all classes which are behaviourally compatible with a given class. The approach, based on relating the external behaviours of classes (as described by a fully-abstract semantics) rather than their individual operations, is used to define behavioural compatibility in Object-Z.

A class B is behaviourally compatible with a class A if objects of B can be substituted for objects of A in any system, or environment in which A can be placed. Therefore, for a class B to be behaviourally compatible with a class A it must be, at least, signature compatible with A. Behavioural compatibility can be defined, therefore, by considering the following two cases.

(1) The class B is signature equivalent with the class A, i.e. an object of B is capable of undergoing exactly the same events as an object of A and, therefore, B can be placed in exactly the same environments as A.

In this case, B will be behaviourally compatible with A if its external behaviours are a subset of the external behaviours of A. If the external behaviours of B are a subset of the external behaviours of A then objects of B will only behave in ways that objects of A could have behaved in the same environment.

If the external behaviours of B are not a subset of the external behaviours of A then there will be a behaviour that an object of B is capable of undergoing which an object of A cannot undergo. Hence, the definition of behavioural compatibility, in this case, is complete.

(2) The class B is not signature equivalent with the class A, i.e. since B is signature compatible with A, an object of B is capable of undergoing any event which an object of A could undergo and, therefore, can be placed in any environment in which an object of A can be placed.

In this case, the information that can be derived about B in any environment in which an object of A could be placed is identical to the information that could be derived about

a class B' which is the same as B but with the operations not common to A removed. B is, therefore, behaviourally compatible with A if B' is behaviourally compatible with A. Since B' will be signature equivalent with A, behavioural compatibility can be determined as in case (1).

This approach can be used to define behavioural compatibility in Object-Z. As a preliminary, the following function *restrict* which restricts a class by removing those operations which are not common to another class with which the original class is signature compatible is defined. (The function *op* is defined as in Section 2.2.1, *sig_compat* as in Section 3.3.2 and *ClassStruct* as in Section 4.3.1.)

$$
\begin{array}{|l}
\mathit{restrict} : \mathit{ClassStruct} \times \mathit{ClassStruct} \pfun \mathit{ClassStruct} \\
\hline
\mathrm{dom}\, \mathit{restrict} = \{(c_2, c_1) : \mathit{ClassStruct} \times \mathit{ClassStruct} \mid c_2 \ \underline{\mathit{sig\_compat}}\ c_1\} \\
\forall (c_2, c_1) : \mathrm{dom}\, \mathit{restrict};\ c_3 : \mathit{ClassStruct} \bullet \\
\quad \mathit{restrict}(c_2, c_1) = c_3 \Leftrightarrow \\
\qquad c_3.\mathit{attr} = c_2.\mathit{attr} \\
\qquad c_3.\mathit{ops} = c_1.\mathit{ops} \\
\qquad c_3.\mathit{op\_params} = c_1.\mathit{ops} \ndres c_2.\mathit{op\_params} \\
\qquad c_3.\mathit{states} = c_2.\mathit{states} \\
\qquad c_3.\mathit{initial} = c_2.\mathit{initial} \\
\qquad c_3.\mathit{trans} = \{e : \mathrm{dom}\, c_2.\mathit{trans} \mid \mathit{op}(e) \in c_1.\mathit{ops}\} \ndres c_2.\mathit{trans} \\
\qquad c_3.\mathit{hist\_inv} = c_2.\mathit{hist\_inv}
\end{array}
$$

Notice that if c_2 $\underline{\mathit{sig\_equiv}}$ c_1, where c_1 and c_2 are the structural models of classes, then $\mathit{restrict}(c_2, c_1) = c_2$.

The external behaviours of a class in Object-Z are given by its complete-readiness model. A class is behaviourally compatible with a given class, therefore, if it is signature compatible with the given class and the behaviours of the class restricted to the operations in the given class is a subset of the behaviours of the given class. This can be formalised using the complete-readiness model $\mathcal{CR}$ of Section 5.3.1 as follows.

$$
\begin{array}{|l}
\mathit{behav\_compat} : \mathit{ClassStruct} \rightarrow \mathit{ClassStruct} \\
\hline
\forall\, c_1, c_2 : \mathit{ClassStruct} \bullet \\
\quad c_2\ \underline{\mathit{behav\_compat}}\ c_1 \Leftrightarrow c_2\ \underline{\mathit{sig\_compat}}\ c_1 \wedge \mathcal{CR}(\mathit{restrict}(c_2, c_1)) \subseteq \mathcal{CR}(c_1)
\end{array}
$$

Notice that a class which has no possible behaviours is behaviourally compatible with any class with which it is signature compatible. Such a class would have no possible initial states and would, hence, be unimplementable.

The definition of *behav_compat* is complete. Given two signature compatible classes with structural models c_1 and c_2, if $\mathcal{CR}(\mathit{restrict}(c_2, c_1))$ is not a subset of $\mathcal{CR}(c_1)$ then there exists a behaviour in $\mathcal{CR}(\mathit{restrict}(c_2, c_1))$ which is not in $\mathcal{CR}(c_1)$ and, therefore, a context C which can distinguish between objects of the classes can be constructed as in Theorem 5.3 of Section 5.3.3. The simplicity of the definition, i.e. the fact that it is simply a subset relation on behaviours, is a direct result of using the fully-abstract model of classes.

Dispenser example

As an example of the use of behavioural compatibility in Object-Z, consider refining the following specification of a dispenser which could be used to deliver chocolates and candy to the customers of a vending machine.

Dispenser

$mode : \{choc, candy\}$

Choc

$\Delta(mode)$

$mode = choc$

Candy

$\Delta(mode)$

$mode = candy$

The class *Dispenser* has a single state variable *mode* representing the next treat to be delivered to a customer and two operations *Choc* and *Candy* representing the delivery of a chocolate and a candy respectively. Initially, and after each operation, the next treat to be delivered is chosen nondeterministically.

A state transition diagram of *Dispenser* is shown in Figure 6.2.

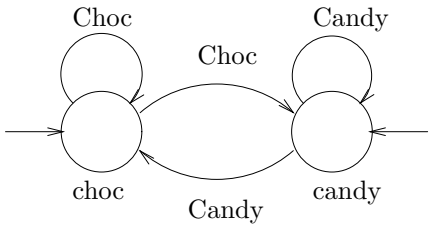


Figure 6.2: State transition diagram of the class *Dispenser*.

The behaviours of the class *Dispenser* consist of all sequences of events corresponding to the operations *Choc* and *Candy* with exactly one event enabled at any time. Therefore, any class which is signature compatible with *Dispenser* and has exactly one event from the set $\{Choc, Candy\}$ enabled at any time will be behaviourally compatible with *Dispenser*.

For example, consider the following class which inherits *Dispenser*.

AltDispenser

Dispenser

Choc

$mode' = candy$

Candy

$mode' = choc$

The operations *Choc* and *Candy* in the class *AltDispenser* are redefined so that the next treat to be delivered after a chocolate is a candy and the next treat to be delivered after a candy is a chocolate.

A state transition diagram of the class *AltDispenser* is shown in Figure 6.3.

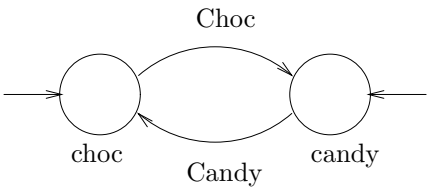


Figure 6.3: State transition diagram of the class *AltDispenser*.

The behaviours of the class *AltDispenser* consist of all alternating sequences of events corresponding to the operations *Choc* and *Candy* with exactly one event enabled at any time. Intuitively, an object of class *AltDispenser* could be substituted for an object of class *Dispenser* in any system.

The type of refinement illustrated above is *procedural refinement* (e.g. see [82]) since only the procedures, i.e. the operations, of the class are refined. As an example of *data refinement* (e.g. see [57]) where the data, i.e. the state, on which the operations are defined is also refined consider the following class.

FairDispenser

$nb\_choc, nb\_candy : \mathbb{N}$

Init

$nb\_choc = nb\_candy = 0$

Choc

$\Delta(nb\_choc)$

$nb\_choc < 2 * nb\_candy$
$nb\_choc' = nb\_choc + 1$

Candy

$\Delta(nb\_candy)$

$nb\_choc \geq 2 * nb\_candy$
$nb\_candy' = nb\_candy + 1$

The class *FairDispenser* has two state variables *nb_choc* representing the number of chocolates already delivered and *nb_candy* representing the number of candies already delivered. The operations *Choc* and *Candy* are defined so that *Choc* occurs when the number of chocolates already delivered is less than double the number of candies already delivered and *Candy* occurs otherwise. The behaviours of the class *FairDispenser*, therefore, consist of sequences of events corresponding to the operations *Choc* and *Candy* with exactly one event enabled at any time. Once again an object of class *FairDispenser* could be substituted for an object of class *Dispenser* in any system.

6.2 Observational and Operational Compatibility

Most object-oriented programming languages regard objects as passive entities which undergo operations only when they are sent messages by the encompassing environment. Such objects can be thought of as being *operated* by their environment. Intuitively, a class will be behaviourally compatible with a given class if an 'operator' of an object of the class cannot deduce that the object is not an object of the given class. This notion of behavioural compatibility is referred to in this thesis as *operational compatibility*.

Some object-oriented programming languages. however, regard objects as active entities which can undergo operations autonomously. A review of such languages as well as a discussion of the use of active objects in object-oriented programming can be found in [47]. Active objects can be thought of as being *observed* by their environment. Intuitively, a class will be behaviourally compatible with a given class if an 'observer' of an object of

the class cannot deduce that the object is not an object of the given class. This notion of behavioural compatibility is referred to in this thesis as *observational compatibility*.

In Object-Z, objects may be regarded as either active or passive allowing the refinement of a specification towards an implementation in a wide variety of programming languages. The definition of behavioural compatibility presented in Section 6.1.2 is strong enough to allow for either situation. Often, however, the language in which a system is to be implemented is known early in the software development process. In such cases, it is desirable to use a weaker notion of behavioural compatibility, based on the particular mode of interaction of an object and its environment in the implementation language, to allow for greater flexibility during refinement.

Section 6.2.1 looks at observational compatibility in Object-Z. This type of compatibility may be used when the specified system is to be implemented in a programming language which supports only active objects. Section 6.2.2 looks at operational compatibility in Object-Z. This type of compatibility may be used when the specified system is to be implemented in a programming language which supports only passive objects. Section 6.2.3 discusses the possibility of integrating observational and operational compatibility. This would be useful if the implementation language supported both active and passive objects or if it supported objects which are partly active (i.e. undergo some operations autonomously) and partly passive (i.e. undergo other operations only when directed to do so by their environment).

6.2.1 Observational compatibility

A class is observationally compatible with a given class if objects of the class cannot be distinguished from objects of the given class by an external observer. The observer is assumed to have no control over the objects which freely undergo any enabled event.

A formal definition of observational compatibility needs to make assumptions about what the observer can and cannot see. This may vary depending on the type of interaction allowed in the language for which the notion of compatibility is being defined. The notion of observational compatibility presented in this section is based on the assumption that an observer can only see the events which an object undergoes. As an example, consider the following Object-Z specification of a bounded queue class which inherits the class $Queue[T]$ of Section 2.1.2.

Let $max : \mathbb{N}$ be a global constant denoting the upper limit on the number of items in any bounded queue.

BoundedQueue[*T*]

Queue[*T*]

$\#items \leqslant max$

Join

$\#items < max$

The class *BoundedQueue*[*T*] adds to *Queue*[*T*] a state invariant which prevents more than *max* items being joined to a queue. An observer who can only see the events an object undergoes could not distinguish an object of class *BoundedQueue*[*T*] from an object of *Queue*[*T*] since each sequence of events that the former class can undergo is also a sequence of events that the latter class can undergo. Therefore, *BoundedQueue*[*T*] may be considered to be observationally compatible with *Queue*[*T*].

This notion of observational compatibility can be formalised for Object-Z using the trace model $\mathcal{T}$ of Section 5.1.2 as follows.

$$obs\_compat : ClassStruct \leftrightarrow ClassStruct$$

$$\forall\, c_1, c_2 : ClassStruct \bullet$$
$$c_2 \ \underline{obs\_compat}\ c_1 \Leftrightarrow c_2 \ \underline{sig\_compat}\ c_1 \wedge \mathcal{T}(restrict(c_2, c_1)) \subseteq \mathcal{T}(c_1)$$

The definition of *obs_compat* is complete. Given two signature compatible classes with structural models c_1 and c_2, if $\mathcal{T}(restrict(c_2, c_1))$ is not a subset of $\mathcal{T}(c_1)$ then there exists a trace which an object of c_2 could undergo in an environment in which an object of c_1 could be placed which an object of c_1 could not undergo in the same environment.

Since $\mathcal{CR}(restrict(c_2, c_1)) \subseteq \mathcal{CR}(c_1) \Rightarrow \mathcal{T}(restrict(c_2, c_1)) \subseteq \mathcal{T}(c_1)$, the relation *obs_compat* is weaker than the relation *behav_compat* of Section 6.1.2. Thus, any two classes which are behaviourally compatible are also observationally compatible.

To use this notion of observational compatibility as a basis for refinement in Object-Z, the environments in which an object could be placed would need to be restricted to reflect an observer's view of an object. That is, references to an object a would be limited to $a.\textsc{Init}$, $a.op$ and $\vec{a}$ where op is an operation in a's class. Since an observer cannot deduce whether or not a particular operation is enabled, the notation pre $a.op$ would not be allowed. Similarly, since an observer cannot force an object to undergo operations, liveness properties which restrict the possible behaviours of a would also not be allowed.

6.2.2 Operational compatibility

A class is operationally compatible with a given class if objects of the class cannot be distinguished from objects of the given class by an external operator. The operator controls the objects by selecting the events they undergo.

A formal definition of operational compatibility needs to make assumptions about which events the operator can select at any stage. The notion of operational compatibility presented in this section is based on the assumption that after the object has undergone any sequence of events, the operator, expecting an object of a particular class, will only select events that an object of that class could not refuse after that sequence of events. If, after a particular sequence of events, an object cannot be guaranteed to be able to undergo any operation then an operator of that object is assumed to make no further selections.

As an example, consider again the classes $Queue[T]$ and $BoundedQueue[T]$. An operator, expecting an object of class $BoundedQueue[T]$, would select *Join* and *Leave* operations with the following restrictions.

- Whenever the difference between the total number of *Join* and *Leave* operations already performed was equal to *max* (i.e. the queue was full), the operator would not select a *Join* operation.
- Whenever the difference between the total number of *Join* and *Leave* operations already performed was zero (i.e. the queue was empty), the operator would not select a *Leave* operation.

If this operator was given an object of class $Queue[T]$ instead of $BoundedQueue[T]$ then the substitution would not be able to be detected as an object of $Queue[T]$ can be operated in the same way. That is, an object of class $Queue[T]$ will not refuse any event that an object of class $BoundedQueue[T]$ is guaranteed to be able to perform after any sequence of events. Therefore, $Queue[T]$ may be considered to be operationally compatible with $BoundedQueue[T]$. Notice that operational compatibility between $Queue[T]$ and $BoundedQueue[T]$ is in the reverse direction to observational compatibility between these classes.

In general, a class B is operationally compatible with a class A if, after any trace of class B which an object of class A can be guaranteed to perform, an object of class B can only refuse those events of class A which an object of class A could have refused. This notion of operational compatibility can be formalised using the definition of *Trace* of Section 5.1.2 and the readiness model $\mathcal{R}$ of Section 5.2.2 as follows.

$$
\begin{array}{|l}
op\_compat : ClassStruct \leftrightarrow ClassStruct \\
\hline
\forall\, c_1, c_2 : ClassStruct \bullet \\
\quad c_2 \ \underline{op\_compat}\ c_1 \Leftrightarrow \\
\qquad c_2 \ \underline{sig\_compat}\ c_1 \\
\qquad \forall\, r_2 : \mathcal{R}(restrict(c_2, c_1)) \bullet \\
\qquad\quad (\bigcap \{r_1 : \mathcal{R}(c_1) \mid r_1.events = r_2.events \bullet r_1.ready\} \subseteq r_2.ready \\
\qquad\quad \lor \\
\qquad\quad \exists\, t : Trace \bullet \\
\qquad\qquad t \subset r_2.events \\
\qquad\qquad r_2.events(\#t + 1) \notin \bigcap \{r_1 : \mathcal{R}(c_1) \mid r_1.events = t \bullet r_1.ready\})
\end{array}
$$

The definition of *op_compat* relies on the assumption that a class has no liveness properties and, hence, there is a ready behaviour corresponding to the trace at each stage in an object's evolution. This is always true for passive objects which cannot force their environment to select operations.

The definition of *op_compat* is complete. Given two signature compatible classes with structural models c_1 and c_2, assume there exists an r in $\mathcal{R}(restrict(c_2, c_1))$ such that an object of c_1 can be guaranteed to perform the trace of r. If there exists an event e such that e is in the intersection of the ready sets of all ready-behaviours in $\mathcal{R}(c_1)$ with the same sequence of events as r but e is not in $r.ready$ then, after performing the sequence of events in r, an object of c_1 would not be able to refuse the event e but an object of c_2 would. Therefore, an operator expecting an object of c_1 would be able to detect that an object of c_2 had been substituted.

The relation *op_compat* is also weaker than the relation *behav_compat* of Section 6.1.2. That is, since $\mathcal{CR}(restrict(c_2, c_1)) \subseteq \mathcal{CR}(c_1) \Rightarrow \mathcal{R}(restrict(c_2, c_1)) \subseteq \mathcal{R}(c_1)$, if indeed $\mathcal{CR}(restrict(c_2, c_1)) \subseteq \mathcal{CR}(c_1)$ then, for all r in $\mathcal{R}(restrict(c_2, c_1))$, r is also in $\mathcal{R}(c_1)$ and, therefore, the intersection of the ready sets of all ready-behaviours in $\mathcal{R}(c_1)$ with the same sequence of events as r must be a subset of the ready set of r. Thus, any two classes which are behaviourally compatible are also operationally compatible.

To use this notion of operational compatibility as a basis for refinement in Object-Z, the environments in which an object could be placed would need to be restricted to reflect the way an operator would control an object. This may vary depending on the possible behaviours of the object which the operator expects. In general, the environment would need to model the operator's understanding of the state of the object so that operations on the object could only occur when they could be selected by the operator. This can be achieved by an auxiliary state variable. For example, consider the following Object-Z context which models a possible operator of the class $BoundedQueue[T]$ of Section 6.2.1.

System[T]

$s : 1 \,..\, max$
$q : \boxdot$

Init

$s = 0$
$q.$*Init*

Join

$\Delta(s)$
$q.$*Join*

$s < max$
$s' = s + 1$

Leave

$\Delta(s)$
$q.$*Leave*

$s > 0$
$s' = s - 1$

The auxiliary state variable s of *System*[T] denotes the number of items in the queue denoted by the object q. The operations *Join* and *Leave* represent the operator performing a *Join* and a *Leave* event on q respectively.

6.2.3 Unifying observational and operational compatibility

Observational compatibility can provide a basis for refinement in Object-Z when the environments in which an object can be placed are restricted to reflect an observer's view of the object. Similarly, operational compatibility can provide a basis for refinement in Object-Z when the environments in which an object can be placed are restricted to reflect the way an operator would control the object. It may be possible, however, that an implementation language allows both active and passive objects or objects which are partly active and partly passive. Specification techniques designed to capture this latter notion of objects have been proposed by Abadi and Lamport[1], Lam and Shankar[69] and Lynch and Tuttle[76].

Adopting the terminology of Lam and Shankar and Lynch and Tuttle, operations under the control of the environment are referred to as *input* operations and those under the control of the object as *output* operations. If the environments in which an object in Object-Z could be placed were limited so that each operation was treated in a way

that both an input and an output operation could be treated then a complete definition of behavioural compatibility could be obtained by conjoining the predicates defining operational and observational compatibility.

If, on the other hand, the environments in which an object could be placed were limited so that particular operations were treated as though they were input operations and others as though they were output operations then a complete definition of behavioural compatibility would not be possible unless input and output operations were explicitly identified in Object-Z.

6.3 Behavioural Compatibility and Inheritance

To reason about classes which contain polymorphic variables (see Section 3.3), it is necessary to consider the behaviours of all classes to which the polymorphic variables can be assigned. In Object-Z, the set of classes to which a polymorphic variable can be assigned are those derived by inheritance from some common class.

Reasoning about classes in Object-Z is simplified, therefore, if an inheritance hierarchy which is to be used polymorphically is restricted so that a given class is behaviourally compatible with the classes it inherits. Under this restriction, a polymorphic variable can only behave in ways that an object of the class at the top of the associated inheritance hierarchy can behave. Section 6.3.1 presents rules for maintaining behavioural compatibility in Object-Z inheritance hierarchies and Section 6.3.2 presents similar rules for maintaining observational and operational compatibility.

6.3.1 Maintaining behavioural compatibility

A class which is behaviourally compatible with a given class must be signature compatible with the given class. Therefore, the rules for maintaining signature compatibility through inheritance must hold if behavioural compatibility is to be maintained. These rules, presented in Section 3.3.2 state that a class must have at least all the operations of any class it inherits and that each redefined operation must have exactly the same parameters as the original inherited operation.

To maintain behavioural compatibility through inheritance, additional rules restricting the redefinition of operations and the addition of initial conditions and history invariants must also hold[2]. To simplify the definition of these rules it will be assumed that no attributes are added to a class during inheritance. The rules could be generalised to allow addition of attributes by introducing a relation between the states of classes along the

[2]Since the state invariant is a conjunct of each operation's precondition and postcondition, the addition of state invariants will be restricted by the rules restricting the redefinition of operations.

lines of the *representation relation* in Hayes[57]. The extra complexity involved, however, is not warranted here.

Given two classes with structural models c_1 and c_2, if c_2 is signature compatible with c_1 then c_2 is also behaviourally compatible with c_1 if the following rules hold. (A proof is given in Theorem 6.1 below.)

Rule 1 - Every initial state of c_2 is an initial state of c_1.

$$c_2.initial \subseteq c_1.initial$$

Rule 2 - For all events e which an object of c_1 can undergo, and all states s, e is enabled in s in c_1 if and only if e is enabled in s in c_2[3].

$$\forall\, e : \mathrm{dom}\, c_1.trans \bullet \\ \quad \mathrm{dom}\, c_1.trans(e) = \mathrm{dom}\, c_2.trans(e)$$

Rule 3 - For all events e which an object of c_1 can undergo, if e is enabled in a state s in both c_1 and c_2 then any state resulting from performing e in s for c_2 can also result from performing e in s for c_1.

$$\forall\, e : \mathrm{dom}\, c_1.trans \bullet \forall\, s : \mathrm{dom}\, c_1.trans(e) \cap \mathrm{dom}\, c_2.trans(e) \bullet \\ \quad c_2.trans(e)(\!|\, \{s\} \,|\!) \subseteq c_1.trans(e)(\!|\, \{s\} \,|\!)$$

Rule 4 - Any total history satisfying the history invariant of c_2 also satisfies the history invariant of c_1.

$$c_2.hist\_inv \subseteq c_1.hist\_inv$$

Rules 1 and 4 are automatically ensured in Object-Z as the initial condition and history invariant of a class can only be strengthened through inheritance.

Rules 2 and 3, however, do not always hold. To maintain behavioural compatibility in an inheritance hierarchy, therefore, a specifier needs to limit the use of redefinition so that the precondition of a redefined operation is the same as the precondition of the inherited operation (to satisfy Rule 2) and the postcondition of a redefined operation is no weaker than the postcondition of the inherited operation (to satisfy Rule 3). While Rule 3 allows the postcondition of an operation to be be strengthened, the possible values that can be assigned to any output parameter in the redefined operation must be the same as the possible values that can be assigned to the same output parameter in the inherited operation. This requirement is necessary as Object-Z does not semantically distinguish

[3]If c_2 is signature compatible with c_1 then an object of c_2 can undergo at least the same events as an object of c_1.

between input and output parameters and allows output parameters to be restricted in an object's environment.

The classes *Dispenser* and *AltDispenser* of Section 6.1.2 satisfy the above rules as the operations in *AltDispenser* are derived from the corresponding operations in *Dispenser* by strengthening their postconditions. Note, however, that not all behaviourally compatible classes need satisfy the rules. For example, the classes *Dispenser* and *FairDispenser* of Section 6.1.2 are behaviourally compatible but are not related by inheritance. A proof that Rules 1, 2, 3 and 4 ensure behavioural compatibility is given below.

Theorem 6.1

Given two classes with structural models c_1 and c_2, if c_2 is signature compatible with c_1 and Rules 1, 2, 3 and 4 hold then c_2 is behaviourally compatible with c_1.

Proof

Since c_2 *sig_compat* c_1, c_2 *behav_compat* c_1 if $\mathcal{CR}(restrict(c_2, c_1)) \subseteq \mathcal{CR}(c_1)$. Since the events enabled in any state are identical for c_1 and c_2 by Rule 2, this will be true if $\mathcal{TH}(restrict(c_2, c_1)) \subseteq \mathcal{TH}(c_1)$. By Rule 4, $c_2.hist\_inv \subseteq c_1.hist\_inv$. Therefore, it is sufficient to prove that $\mathcal{TH}_{safe}(restrict(c_2, c_1)) \subseteq \mathcal{TH}_{safe}(c_1)$.

The proof is by induction over the length of $h.events$ where $h \in \mathcal{TH}_{safe}(restrict(c_2, c_1))$.

(i) If $\#h.events = 0$ then the state of h will be in $c_2.initial$. By Rule 1, this state is also in $c_1.initial$. Therefore, h is in $\mathcal{TH}_{safe}(c_1)$.

(ii) Given $n \geqslant 0$, assume for all h such that $\#h.events = n$, $h \in \mathcal{TH}_{safe}(c_1)$.

Since $\mathcal{TH}_{safe}(restrict(c_2, c_1))$ contains all pre-histories of its histories, if $\#h.events = n+1$ then there exists a history h_0 in $\mathcal{TH}_{safe}(restrict(c_2, c_1))$, where $h_0.events = front\ h.events$ and $h_0.states = front\ h.states$. By the assumption, h_0 is also in $\mathcal{TH}_{safe}(c_1)$ and, by Rule 2, the event $h.events(n+1)$ is enabled in the final state of h_0 for c_1. Furthermore, by Rule 3, $h.states(n+2)$ is a possible post-state of $h.events(n+1)$ in c_1 when the pre-state is $h.states(n+1)$, i.e. $last\ h_0.states$. Therefore, h is in $\mathcal{TH}_{safe}(c_1)$. □

6.3.2 Maintaining observational and operational compatibility

In this section, rules for maintaining observational and operational compatibility through inheritance are presented. These rules can be used to restrict inheritance in a hierarchy of classes representing either active or passive objects. As in Section 6.3.1, it will be assumed the attributes of a class and the classes it inherits are identical.

Observational compatibility

Since observational compatibility is weaker than behavioural compatibility, given two classes with structural models c_1 and c_2, c_2 will be observationally compatible with c_1 if it is signature compatible with c_1 and Rules 1, 2, 3 and 4 of Section 6.3.1 hold. It is possible, however, to strengthen Rule 2 as follows.

Rule 2′ - For all events e which an object of c_1 can undergo, and all states s, if e is enabled in s in c_2 then e is enabled in s in c_1.

$$\forall\, e : \mathrm{dom}\, c_1.trans \bullet \\ \quad \mathrm{dom}\, c_2.trans(e) \subseteq \mathrm{dom}\, c_1.trans(e)$$

This rule allows the specifier to strengthen the precondition of an inherited operation. In combination with Rule 3, it also allows the possible values that can be assigned to any input or output parameter in a redefined operation to be a subset of the possible values that can be assigned to the same parameter in the inherited operation. The class $Queue[T]$ of Section 2.1.2 and the class $BoundedQueue[T]$ of Section 6.2.1 satisfy the above rule as the operations in $BoundedQueue[T]$ are derived from the corresponding operations in $Queue[T]$ by strengthening their preconditions. A proof that Rules 1, 2′, 3 and 4 ensure observational compatibility is given below.

Theorem 6.2

Given two classes with structural models c_1 and c_2, if c_2 is signature compatible with c_1 and Rules 1, 2′, 3 and 4 hold then c_2 is observationally compatible with c_1.

Proof

Since c_2 $\underline{sig\_compat}$ c_1, c_2 $\underline{obs\_compat}$ c_1 if $\mathcal{T}(restrict(c_2, c_1)) \subseteq \mathcal{T}(c_1)$. This will be true if $\overline{\mathcal{TH}(restrict(c_2, c_1))} \subseteq \overline{\mathcal{TH}(c_1)}$. By Rule 4, $c_2.hist\_inv \subseteq c_1.hist\_inv$. Therefore, it is sufficient to prove that $\mathcal{TH}_{safe}(restrict(c_2, c_1)) \subseteq \mathcal{TH}_{safe}(c_1)$.

The proof is similar to that in Theorem 6.1 of Section 6.3.1 except that Rule 2′ replaces Rule 2. □

The above relation on operations is referred to as a *covariance* relation as the precondition and the postcondition of an operation may be changed in the same way, i.e. they can both be strengthened. Most existing approaches to behavioural compatibility (e.g. [8, 114, 119]), however, suggest a contravariance relation where the precondition can be weakened and the postcondition strengthened. The discrepancy arises because these approaches assume that an object is a passive, rather than an active, entity which only behaves correctly when its environment operates it by offering it operations which it can be guaranteed to perform. The existing approaches are, therefore, similar to the notion of operational compatibility defined in Section 6.2.2.

Operational compatibility

Since operational compatibility is weaker than behavioural compatibility, given two classes with structural models c_1 and c_2, c_2 will be operationally compatible with c_1 if it is signature compatible with c_1 and Rules 1, 2, 3 and 4 of Section 6.3.1 hold[4]. It is possible, however, to strengthen Rule 2 as follows.

Rule 2″ - For all events e which an object of c_1 can undergo, and all states s, if e is enabled in s in c_1 then e is enabled in s in c_2.

$$\forall\, e : \mathrm{dom}\, c_1.trans \bullet \\ \quad \mathrm{dom}\, c_1.trans(e) \subseteq \mathrm{dom}\, c_2.trans(e)$$

This rule allows the specifier to weaken the precondition of an inherited operation. In combination with Rule 3, the relation on operations is, therefore, a contravariant relation agreeing with the existing approaches to behavioural compatibility. The class $BoundedQueue[T]$ of Section 6.2.1 and the class $Queue[T]$ of Section 2.1.2 satisfy the above rule as the operations in $Queue[T]$ could be derived from the corresponding operations in $BoundedQueue[T]$ by weakening their preconditions. A proof that Rules 1, 2″, 3 and 4 ensure operational compatibility is given below.

Theorem 6.3

Given two classes with structural models c_1 and c_2, if c_2 is signature compatible with c_1 and Rules 1, 2″, 3 and 4 hold then c_2 is operationally compatible with c_1.

Proof

Since c_2 $\underline{sig\_compat}$ c_1, c_2 $\underline{obs\_compat}$ c_1 if the following is true.

$$\forall\, r_2 : \mathcal{R}(restrict(c_2, c_1)) \bullet \\ \quad (\bigcap\{r_1 : \mathcal{R}(c_1) \mid r_1.events = r_2.events \bullet r_1.ready\} \subseteq r_2.ready \\ \quad \vee \\ \quad \exists\, t : Trace \bullet \\ \quad\quad t \subset r_2.events \\ \quad\quad r_2.events(\#t + 1) \notin \bigcap\{r_1 : \mathcal{R}(c_1) \mid r_1.events = t \bullet r_1.ready\})$$

By Rule 2″, the events enabled in any state of c_1 are a subset of the events enabled in the same state in c_2. Therefore, the above relationship will be true if, for all histories h in $\mathcal{TH}(restrict(c_2, c_1))$, either

(1) h is in $\mathcal{TH}(c_1)$ or

[4]The history invariant in this case is assumed to consist of only safety, and not liveness, properties.

(2) there exists a history h_1 in $\mathcal{TH}(c_1)$ such that h_1 is a pre-history of h and the event $h.events(\#h_1.events + 1)$ is not enabled in $last\, h_1.states$.

By Rule 4, $c_2.hist\_inv \subseteq c_1.hist\_inv$. Therefore, it is sufficient to prove that, for all histories h in $\mathcal{TH}_{safe}(restrict(c_2, c_1))$, either

(1) h is in $\mathcal{TH}_{safe}(c_1)$ or

(2) there exists a history h_1 in $\mathcal{TH}_{safe}(c_1)$ such that h_1 is a pre-history of h and the event $h.events(\#h_1.events + 1)$ is not enabled in $last\, h_1.states$.

The proof is by induction over the length of $h.events$ where $h \in \mathcal{TH}_{safe}(restrict(c_2, c_1))$.

(i) If $\#h.events = 0$ then the state of h will be in $c_2.initial$. By Rule 1, this state is also in $c_1.initial$. Therefore, h is in $\mathcal{TH}_{safe}(c_1)$ and (1) is true.

(ii) Given $n \geqslant 0$, assume for all h such that $\#h.events = n$, either (1) or (2) is true.

Since $\mathcal{TH}_{safe}(restrict(c_2, c_1))$ contains all pre-histories of its histories, if $\#h.events = n+1$ then there exists a history h_0 in $\mathcal{TH}_{safe}(restrict(c_2, c_1))$, where $h_0.events = front\, h.events$ and $h_0.states = front\, h.states$. By the assumption, h_0 is either in $\mathcal{TH}_{safe}(c_1)$ or there exists a history h_1 in $\mathcal{TH}_{safe}(c_1)$ such that h_1 is a pre-history of h_0 and $h_0.events(\#h_1.events+1)$ is not enabled in $last\, h_1.events$.

If h_0 is in $\mathcal{TH}_{safe}(c_1)$ and the event $h.events(n + 1)$ is enabled in the final state of h_0 for c_1 then, by Rule 3, $h.states(n + 2)$ is a possible post-state of $h.events(n + 1)$ in c_1 when the pre-state is $h.states(n + 1)$, i.e. $last\, h_0.states$. Therefore, h is in $\mathcal{TH}_{safe}(c_1)$ and (1) is true.

If h_0 is in $\mathcal{TH}_{safe}(c_1)$ and the event $h.events(n + 1)$ is not enabled in the final state of h_0 for c_1 then, since h_0 is a pre-history of h, (2) is true.

If there exists a history h_1 in $\mathcal{TH}_{safe}(c_1)$ such that h_1 is a pre-history of h_0 and the event $h_0.events(\#h_1.events + 1)$ is not enabled in $last\, h_1.states$ then, since h_0 and h_1 are pre-histories of h, (2) is true. □

Chapter 7

Conclusions

"Action will remove the doubt that theory cannot solve."

— Tehyi Hsieh
Chinese Epigrams Inside Out and Proverbs, 1948.

Formal methods for software development are becoming increasingly necessary as software becomes an important part of everyday life. To handle the complexities inherent in large-scale software systems these methods need to be combined with a sound development methodology which supports modularity and reusability. Object orientation, based on the concept that systems are composed of collections of interacting objects whose behaviours are specified by classes, is such a methodology.

This thesis has presented the formal specification language Object-Z which is an extension of the formal specification language Z to facilitate specification in an object-oriented style. The major extension in Object-Z is the introduction of the *class schema* which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are objects. Object-Z also supports single and multiple inheritance allowing classes to be reused in the definition of other classes and polymorphism allowing a variable to be assigned to objects of more than one class.

The thesis has also presented a set-theoretic model of classes in Object-Z which could form the basis of a full formal semantics. The model, based on the *histories* of a class, i.e. the sequences of states and operations which an object of the class can undergo, facilitates the specification of liveness properties using a temporal logic notation. A fully-abstract model of classes in Object-Z, derived from the history model, was also presented. This model was used to formally define a notion of behavioural compatibility in Object-Z which could form the basis of a theory of class refinement.

While other versions of Object-Z have already had some application outside academia[103, 43], before the language can be successfully applied to each stage of the development of large-scale software systems, a complete formal semantics and theory of refinement need to be completed and integrated with the existing semantics and refinement rules of Z. This will enable the development of semi-automatic tools to aid in the processes of specification, verification and refinement. Section 7.1 presents a summary of the work presented in this thesis and Section 7.2 discusses the contributions with respect to related work. Section 7.3 indicates future research directions including possible extensions to the Object-Z language.

7.1 Thesis Summary

Object-Z is an extension of Z in that the full syntax and semantics of Z are retained. Hence, any Z specification is also an Object-Z specification. The extensions in Object-Z support both an object-oriented style of specification and the specification of liveness properties such as fairness, termination and the guaranteed occurrence of operations.

Chapter 2 of this thesis introduced the notion of class schemas in Object-Z. A class schema, or class, encapsulates a single state schema with its initial state schema and all the operations which can affect its variables. This encapsulation, by explicitly grouping related schemas, improves both the clarity and opportunity for reuse of specifications.

The meaning of a class is a set of values corresponding to potential objects of the class at some stage of their evolution. The value chosen to represent an object is the object's history, i.e. the sequence of states it has passed through together with the sequence of events it has undergone.

A class, therefore, defines a type. An object may be declared as an instance of a class within another class enabling the specification of composite objects or systems containing aggregates of objects of the same class. The state of an object is hidden from its environment which may only initialise the object or apply operations to it.

Chapter 3 presented inheritance in Object-Z. Through inheritance, new classes can be specified as extensions or specialisations of one or more existing classes without the need to re-specify the common state variables or operations. When a class inherits another class in Object-Z, the schema definitions of the classes are merged. A class which inherits a given class may, therefore, have additional state variables and operations and additional constraints on the state and the precondition and postcondition of each operation.

A class may, however, also rename any state variable or operation it inherits and arbitrarily redefine any operation. Renaming allows name clashes to be avoided during multiple inheritance and also allows more meaningful or appropriate names to be given to state variables and operations when a class is specialised for a particular application. Redefinition allows the specification of classes which share the structure, but not the behaviour, of a given class.

Inheritance in Object-Z also provides the basis for polymorphism. A variable may be declared to be an object of a particular class or any class derived from that class by inheritance. If a variable is to be used polymorphically then this must be explicitly declared, i.e. not all object-valued variables are polymorphic.

A variable which is declared to be polymorphic can only be placed in an environment in which an object of any class within the associated inheritance hierarchy can be placed. The responsibility of the specifier to ensure this can be reduced if inheritance is restricted so that a given class is signature compatible with the classes it inherits. Rules for maintaining signature compatibility through inheritance were presented in Section 3.3.2.

Chapter 4 discussed the specification of liveness properties in Object-Z. A full formal syntax and semantics of a temporal logic notation was presented which enables the specification of liveness properties concerned with the occurrence of both states and events. This notation can be used in Object-Z classes to specify *history invariants* which restrict the set of histories derived from the state and operations of the class.

When an object of a class with a liveness property is instantiated within another class, the value of the object is, at any time, a history satisfying the safety property of its class. A history invariant of the class in which it is instantiated ensures that the object's history progresses until it also satisfies the liveness property of its class.

Chapter 5 presented a fully-abstract model of classes in Object-Z. This model contains the minimum amount of information required to enable the denotation of a class to be derived from the denotations of the classes of the objects of which it is composed. Intuitively, the model describes the external behaviour of a class, and, hence, captures the precise meaning of a class independent of its syntactic representation.

The model, called the *complete-readiness* model, represents an object by the sequence of events it has undergone together with the sequence of sets of events representing the enabled events at each stage of its evolution. The model was proved to be fully-abstract with respect to an observational model of classes which represents objects by the sequence of events, or *traces*, they have undergone.

Chapter 6 used the fully-abstract model of Chapter 5 to define a notion of behavioural compatibility in Object-Z. This definition could be used as the basis for a theory of class refinement enabling specifications to be refined by separately refining the classes of each of their component objects. The definition is also complete allowing all classes which are behaviourally compatible with a given class to be identified.

Notions of *observational* and *operational* compatibility were also defined. Observational compatibility is relevant when an object is assumed to be an active entity which undergoes events autonomously. Operational compatibility, on the other hand, is relevant when an object is assumed to be a passive entity which is controlled by its environment. These weaker notions of behavioural compatibility could form the basis of a theory of class refinement when the environments in which an object can be placed are restricted to reflect the appropriate mode of object/environment interaction.

Rules for maintaining behavioural compatibility through inheritance were also presented. These rules restrict the way in which operations can be redefined. General behavioural compatibility, holds if each redefined operation has an identical precondition with the inherited operation and a postcondition which is possibly stronger than that of the inherited operation. Observational compatibility allows a covariant redefinition rule where both the precondition and postcondition of a redefined operation may be strengthened, and operational compatibility a contravariant redefinition rule where the precondition of a redefined operation may be weakened and the postcondition strengthened.

7.2 Related Work

In this section, the work in this thesis is compared to related work in the field. In particular, other approaches to object-oriented Z, modelling objects and behavioural equivalence and compatibility are examined.

7.2.1 Object-oriented Z

Object-Z is not the only object-oriented adaptation of Z. It has, however, been developed independently of all other approaches. Some of these other approaches have been reviewed in Section 1.3.2 and will be briefly compared to Object-Z here. A more comprehensive comparative study of object orientation in Z can be found in [110].

According to Wegner[116], a language is object-oriented if it supports the notions of objects, classes and inheritance. Object-Z supports all these notions as well as a notion of polymorphism and an ability to specify liveness properties. Other approaches to object orientation in Z, however, do not support the three basic concepts. The approach of Hall[55] does not extend Z and so has no explicit notion of class or inheritance. Also, the approach of Schuman and Pitt[98, 99] only approximates the notion of a class with naming conventions and has no way of implicitly inheriting all the operations along with the state schema of a class.

The approaches of Whysall and McDermid[118, 119] and Lano[73], while supporting the fundamental concepts of object orientation, are less general, and arguably more difficult to use, than Object-Z as they have been designed with a particular purpose in mind. The approach of Whysall and McDermid was designed specifically to aid the process of refinement. It requires each class to be specified both in standard Z and as an algebraic export specification. The approach of Lano was developed as part of the ESPRIT REDO project on reverse-engineering. It is particularly concerned with separating the implementational details of data structures from the high-level details of system functionality. A class is not specified using a Z-like notation but using constructs similar to those found in object-oriented programming languages.

The approach of Cusack[35, 34], being based on Object-Z, is very similar to it. It is the only other approach which currently supports a notion of polymorphism. This notion is quite different to that in Object-Z, however, as it is based on subtype, rather than subclass (or inheritance), hierarchies and object-valued variables are always treated polymorphically. While the notions of inheritance and polymorphism have been formally defined for this approach, many other aspects, including the formal syntax of classes, are far less developed than in Object-Z.

The final approach reviewed in Section 1.3.2 was the OOZE approach of Alencar and Goguen[4]. This approach is similar syntactically to Object-Z but has the advantage that it has a full formal semantics as it adopts the semantics of the object-oriented programming language FOOPS[50]. This approach, however, is still in an early stage of development and has not been successfully applied to any significant case studies.

Object-Z is the only object-oriented adaptation of Z which currently allows the specification of liveness properties. While the ability to specify such properties is not important when modelling systems of passive objects, it is important in the specification of active object systems. The ability to specify liveness properties allows the specifier to abstract away from implementational mechanisms which ensure those liveness properties hold.

7.2.2 Modelling objects

Two main approaches to modelling objects are found in the literature. The first is based on modelling an object in terms of its (implementational) structure. This approach, adopted by Wolczko[122], Kamin[68] and Reddy[94] among others, models objects as records whose fields are state variables and operations. The structural model of Section 2.2.1 is such a model. While these models have an intuitive appeal, they are not very abstract and are not suited to modelling general liveness properties of objects.

The second approach is based on modelling an object in terms of its behaviour. This approach, adopted by Goguen[49] and Ehrich and Sernadas[45] among others, models objects by the sequences of states and/or sequences of events that they undergo. The history model of Section 2.2.2 and the complete-readiness model of Section 5.3.1 are such models. These models are, in general, more abstract as they represent a canonical form of the object in which unreachable states and state transitions are ignored. This is necessary for a complete notion of behavioural compatibility to be defined as was seen in Section 6.1.1. General liveness properties can also be captured by restricting the sequences of states and/or events which an object can undergo.

These two approaches are not peculiar to modelling objects. They have also been used as the basis of the semantics of process algebras and to model modules in techniques for specifying concurrent systems. The semantics of CCS[81] is based on the structural approach modelling a process in terms of its possible states and transitions. The semantics of CSP[61], on the other hand, models a process in terms of the sequences of events it

can undergo and the possible sets of events it can refuse at any time. Similarly, modules are specified in terms of states and transitions by Lam and Shankar[69] and Lynch and Tuttle[76] and in terms of sequences of states by Abadi and Lamport[1].

7.2.3 Behavioural equivalence

The fully-abstract model of classes in Section 5.3.1 defines the notion of behavioural equivalence of classes in Object-Z. That is, two classes are behaviourally equivalent when they have exactly the same complete-readiness model. This was shown to be a stronger form of behavioural equivalence than that of CSP which is based on processes having the same failures model. This enables Object-Z to capture notions of priority and alternative notions of composition which cannot be captured in CSP.

The notion of behavioural equivalence is, however, weaker than *bisimulation* which is the primitive notion of equivalence between processes in CCS. To detect that two processes are not bisimilar, it may be necessary for the environment to make multiple copies of a process at some stage of its evolution and to 'force' the copies to undergo every possible transition which the process could have undergone as its next transition. For example, the processes P and Q in Figure 7.1 are not bisimilar. They are, however, equivalent in terms of the complete-readiness model.

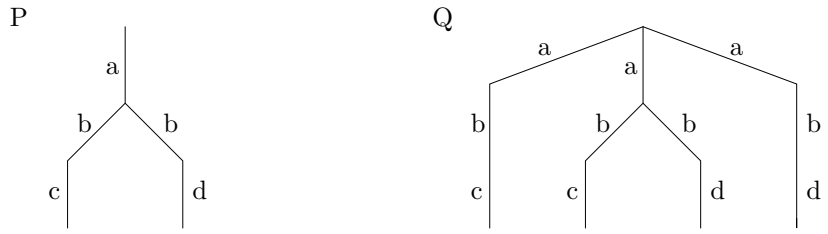


Figure 7.1: Processes P and Q.

Whether bisimulation provides an intuitive notion of behavioural equivalence has been the subject of some debate. The notion of 'forcing' a process to exhibit every possible transition has been criticised by Bloom *et al.*[16] as it assumes the environment of a process can control its internal nondeterminism. On the other hand, Larsen and Skou[74] argue that non-bisimilar processes can be detected using a notion of probabilistic testing.

An alternative notion of behavioural equivalence is proposed by Bloom *et al.* which allows multiple copies to be made of processes but does not allow the internal nondeterminism of a process to be controlled. Bloom and Meyer[17] argue that this equivalence is the finest "reasonable" process equivalence. The notion of behavioural equivalence in terms of the complete-readiness model is, however, still weaker than this equivalence. For example, consider the processes R and S in Figure 7.2.

These processes are not equivalent according to the notion of behavioural equivalence of Bloom *et al.* but are equivalent in terms of the complete-readiness model. The discrepancy

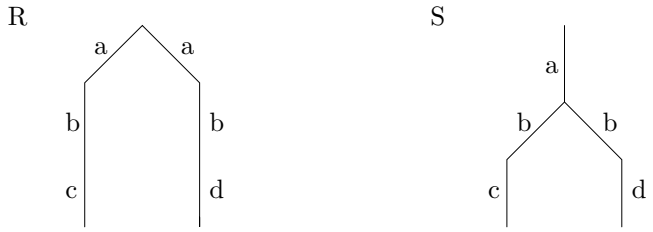
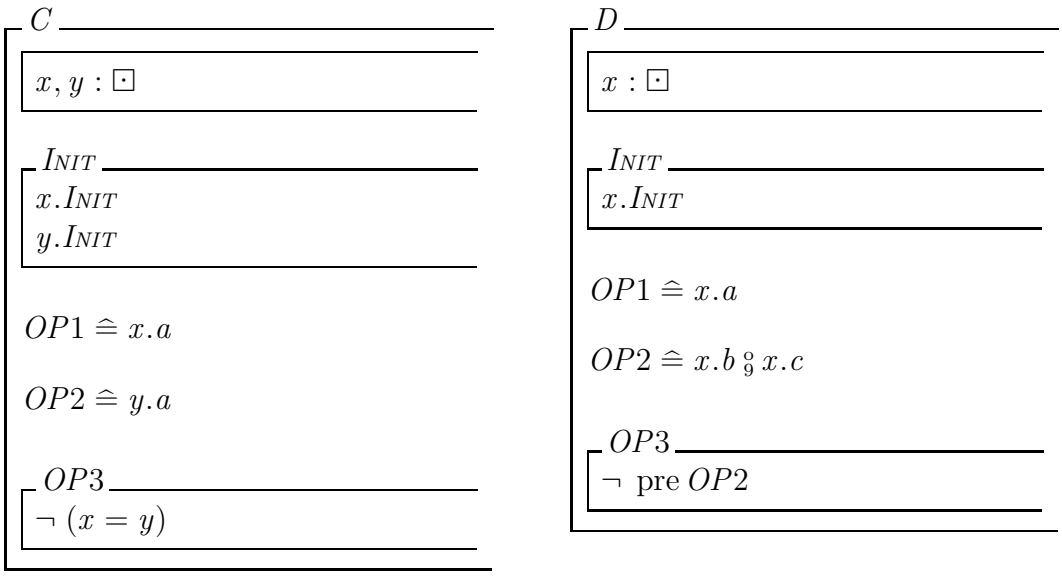


Figure 7.2: Processes R and S.

arises because the environment of an object in Object-Z cannot make copies of the object. Allowing the environment in Object-Z to do this would violate the notion that an object's state is hidden.

Therefore, classes corresponding to the processes R and S need to be semantically identified in a fully-abstract semantics of Object-Z. This is precisely the reason why expressions such as $a = b$ and $a \in A$, where a and b are objects and A is a set of objects, are not allowed in Object-Z. It is also the reason why schema expressions involving the Z schema operator $\fatsemi$ are not allowed in Object-Z classes. Allowing such expressions enables the following contexts, which distinguish classes corresponding to the processes R and S, to be constructed.



When an object of a class corresponding to the process R is placed in the context C the trace $\langle(\text{`}OP1\text{'}, \varnothing), (\text{`}OP2\text{'}, \varnothing), (\text{`}OP3\text{'}, \varnothing)\rangle$ is possible[1]. This trace is not possible, however, when a class corresponding to the process S is placed in C.

Similarly, the trace $\langle(\text{`}OP1\text{'}, \varnothing), (\text{`}OP3\text{'}, \varnothing)\rangle$ is possible when a class corresponding to the

[1]Since $OP1$, $OP2$ and $OP3$ have no parameters, the associated assignment of values is denoted by the empty set.

process R is placed in the context D but not when a class corresponding to the process S is placed in D.

7.2.4 Behavioural compatibility

Most existing definitions of behavioural compatibility in the literature on object orientation are based on the contravariance approach reviewed in Section 6.1.1. It was shown that such definitions are not complete and, hence, cannot necessarily be used to identify all classes which are behaviourally compatible with a given class. Furthermore, it was shown that while the contravariance approach is appropriate for passive objects which are operated by their environment, a covariance approach is required for active objects.

Alternative approaches to behavioural compatibility can be found in the literature on process algebras. In general, these approaches assume a process is passive and operated by its environment. Of particular interest is the notion of *conformance*[21]. A process P is said to conform to a process Q if P can be operated as if it were Q and not deadlock unless Q could have also deadlocked. This notion is similar to the notion of operational compatibility defined in Section 6.2.2. However, in the case of conformance the operator can select any sequence of events that the expected process can undergo even if, due to nondeterminism, the process may deadlock. In the case of operational compatibility, the operator can only select those sequences of events which can be guaranteed not to deadlock. As was shown in Section 6.3.2, this view of an operator allows operational compatibility to capture a similar notion of behavioural compatibility as the widely accepted contravariance approach.

7.3 Future Work

In this section, possible areas of future work are described. In particular, possible extensions to the Object-Z language are discussed. One such extension is the introduction of the notion of *internal* operations which are hidden from an object's environment. Such operations may correspond to the interactions between the components of a composite object, e.g. the operation *Transfer* of the class *Channel* in Section 2.3.1. Explicitly declaring such operations to be internal would prevent them being further constrained by the environment of the composite object.

The notion of internal operations has been widely used (e.g. [61, 81, 1, 69, 76]). The notion could be incorporated syntactically into Object-Z using a list similar to the redefine list discussed in Section 3.2.2. Indeed an earlier version of Object-Z[27] included such a list. Semantically, the notion could be incorporated by removing from the histories of a class those transitions corresponding to internal operations. This approach is similar to that adopted in CSP.

Another possible extension is the explicit identification of *input* and *output* operations which are controlled by the environment of an object and the object itself respectively.

This would enable a specifier to verify that a particular specification is realisable using a method similar to that of Abadi and Lamport[1]. It would also enable a complete definition of behavioural compatibility for classes when the environments in which an object could be placed were limited so that particular operations were regarded as input operations and others as output operations.

The notion of input and output operations have been discussed by Lam and Shankar[69] and Lynch and Tuttle[76] and a similar notion of *agents* by Abadi and Lamport[1]. The notion could also be incorporated syntactically into Object-Z using input and output lists. The possibility of making an operation's inclusion in an input or output list optional would enable a specifier to abstract away from the notion of how the operation is controlled. Such an operation could, at some later stage, be refined to be either an input or an output operation.

Object-Z introduces a new type of schema, namely the class schema, but does not introduce any operators for these schemas apart from inclusion, i.e. inheritance. Other operators, possibly based on the schema operators of Z, could be introduced allowing more flexible means of incrementally modifying and combining classes. It may also be possible to introduce operators between objects rather than their operations. In particular, a concurrency operator similar to those found in process algebras could be defined.

Before Object-Z can be used in the development of software a full formal semantics mapping constructs in the language to some semantic domain needs to be defined. This would enable the development of a proof system for the language and the possibility of semi-automatic tools to aid in the software development process. The history model of classes presented in Section 2.2.2 could form the basis of such a semantics.

Since Object-Z is an extension to Z, a natural approach to developing its formal semantics would be to extend the formal semantics of Z[107]. Such an approach has been adopted by Duke and Duke in [39]. The abstract syntax of types in Z is extended to include a class type and a 'class variety' similar to the notion of a schema variety in [107] is introduced.

Finally, an important area of future work is the application of Object-Z to the development of real systems. Only in this way can the expected benefits of object orientation in the development of such systems be ascertained. Such application would also highlight the need, or otherwise, of the proposed extensions to the language and, possibly, suggest other more useful extensions and conventions.

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, 1990.

[2] J.-R. Abrial, S. Schuman, and B. Meyer. Specification language. In R.M. McKeag and A. Macnaghten, editors, *On the Construction of Programs: An advanced course*, pages 343–410. Cambridge University Press, 1980.

[3] H. Aït-Kaci. Type subsumption as a model of computation. In *Expert Database Systems*. Benjammin/Cummings Inc, 1986.

[4] A. Alencar and J. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *Proceedings European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 1991.

[5] J. Almarode. Rule-based delegation for prototypes. In *Proceedings 4th ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, pages 363–370, 1989.

[6] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[7] P. America. Inheritance and subtyping in a parallel object-oriented language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[8] P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Eindhoven, The Netherlands, 1989.

[9] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[10] D. Anderson and L. Landweber. Protocol specification by Real-Time Attribute Grammars (extended abstract). In Y. Yemini, R. Strom, and S. Yemini, editors, *Protocol Specification, Testing, and Verification, IV*, pages 457–465. North-Holland, 1985.

[11] ANSI and AJPO. *Military Standard: Ada Programming Language*. Am. Nat. Standards Inst. and US Gov. Department of Defense, Ada Joint Program Office, 1983. ANSI/MIL-STD-1815A-1983.

[12] F. Belina and D. Hogrefe. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, March 1989.

[13] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[14] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Auerbach, 1973.

[15] S. Black. Objects and LOTOS. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*. North-Holland, 1990.

[16] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced: Preliminary report. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988.

[17] B. Bloom and A. Meyer. Experimenting with process equivalence. In M. Kwiatkowska, M. Shields, and R. Thomas, editors, *Proceedings BCS-FACS Workshop on Semantics for Concurrency*, Workshops in Computing, pages 81–95. Springer-Verlag, 1990.

[18] D. Bobrow and M. Stefik. *LOOPS: an Object-Oriented Programming System for Interlisp*. Xerox PARC, 1982.

[19] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[20] G. Booch. *Object-Oriented Design with Applications*. Addison-Wesley, 1990.

[21] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In B. Sarikaya and G. von Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*. North-Holland, 1987.

[22] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(7):560–599, 1984.

[23] R. Burstall and J. Goguen. An informal introduction to specifications using Clear. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, International Lecture Series in Computer Science, chapter 4, pages 185–213. Academic Press, 1981.

[24] H. Cannon. Flavors. Technical report, MIT Artificial Intelligence Laboratory, 1980.

[25] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[26] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computer Surv.*, 17(4):471–522, December 1985.

[27] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. North-Holland, 1990.

[28] S. Clerici and F. Orejas. GSBL: An algebraic specification language based on inheritance. In S. Gjessing and K. Nygaard, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 1988.

[29] W. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[30] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. Technical Report STL-89-17, Hewlett-Packard Labs, 1989.

[31] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings 4th ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, pages 433–444, 1989.

[32] B. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[33] E. Cusack. Refinement, conformance and inheritance. In *BCS Workshop on Refinement.* Open University, 1989.

[34] E. Cusack. Inheritance in object oriented Z. In P. America, editor, *Proceedings European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 167–179. Springer-Verlag, 1991.

[35] E. Cusack and M. Lai. Object oriented specification in LOTOS and Z or, my cat really is object oriented! In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 179–202. Springer-Verlag, 1991.

[36] E. Cusack, S. Rudkin, and C. Smith. An object oriented interpretation of LOTOS. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 211–226. North-Holland, 1990.

[37] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computer Surv.*, 20(1):29–72, March 1988.

[38] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[39] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 1990.

[40] R. Duke, I. Hayes, P. King, and G. Rose. Protocol specification and verification using Z. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification, VIII*, pages 33–46. North-Holland, 1988.

[41] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall International, 1991.

[42] R. Duke, G. Rose, and G. Smith. Object-oriented formal specification: Case studies. Unpublished manuscript, 1989.

[43] R. Duke, G. Rose, and G. Smith. Transferring formal techniques to industry. In J. Quemada, J. Mañas, and E. Vazquez, editors, *Formal Description Techniques, III (FORTE'90)*, pages 279–286. North-Holland, 1990.

[44] H. Ehrich, J. Goguen, and A. Sernadas. A categorical theory of objects as observed proceedingssses. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer-Verlag, 1991.

[45] H. Ehrich and A. Sernadas. Algebraic implementation of objects over objects. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 239–266. Springer-Verlag, 1990.

[46] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[47] C. Ellis and S. Gibbs. Active objects: Realities and possibilities. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 22, pages 561–572. ACM Press, 1989.

[48] C. Fidge. A formal definition of priority in CSP. In *Proceedings 15th Australian Computer Science Conference (ACSC-15)*, pages 267–284, January 1992.

[49] J. Goguen. Sheaf semantics for concurrent interacting objects. In *Proceedings REX School/Workshop on Foundations of Object-Oriented Languages*, 1990.

[50] J. Goguen and J. Meseguer. Unifying functional, object-oriented, and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[51] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, pages 391–420. Addison-Wesley, 1985.

[52] J. Goguen and D. Wolfram. On types and FOOPS. In *Proceedings IFIP TC-2 Working Conference on Database Semantics*, 1990.

[53] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[54] D. Halbert and P. O'Brien. Using types and inheritance in object-oriented languages. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 20–31. Springer-Verlag, 1987.

[55] A. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. Springer-Verlag, 1990.

[56] I. Hayes, editor. *Specification Case Studies*. Series in Computer Science. Prentice-Hall International, 1987.

[57] I. Hayes. Bias in VDM: Full abstraction and the functional retrieve rules for data refinement. Technical Report 162, Department of Computer Science, University of Queensland, Australia, May 1990.

[58] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–339, November 1989.

[59] I. Hayes, M. Mowbray, and G. Rose. Signalling system No.7: The network layer. In *Protocol Specification, Testing, and Verification, IX*. North-Holland, 1989.

[60] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[61] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[62] D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Trans. Software Engineering*, 14(9):1243–1252, September 1988.

[63] ISO TC97/SC21. *Estelle – A Formal Description Technique Based on an Extended State Transition Model*, 1988. International Standard 9074.

[64] ISO TC97/SC21. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988. International Standard 8807.

[65] C. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1986.

[66] G. Jones and M. Goldsmith. *Programming in occam 2*. Series in Computer Science. Prentice-Hall International, 1988.

[67] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings 16th ACM Symposium on Principles of Programming Languages*, pages 155–165, 1989.

[68] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[69] S. Lam and A.U. Shankar. Understanding interfaces. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV (FORTE'91)*, pages 165–184, 1991.

[70] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engineering*, 3(2):125–143, 1977.

[71] L. Lamport. An axiomatic semantics of concurrent programming languages. In K. Apt, editor, *Proceedings NATO Advanced Course on Logics and Models of Concurrent Systems*, pages 77–122. Springer-Verlag, 1985.

[72] L. Lamport. A temporal logic of actions. Technical Report 57, Digital Systems Research Centre, April 1990.

[73] K. Lano. Z++. In J.E. Nicholls, editor, *Z User Workshop Oxford 1990*. Springer-Verlag, 1990.

[74] K. Larsen and A. Skou. Bisimulation through probabilistic testing (preliminary report). In *Proceedings 16th ACM Symposium on Principles of Programming Languages*, pages 344–352, 1989.

[75] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[76] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[77] T. Mayr. Specification of object-oriented systems in LOTOS. In K. Turner, editor, *Formal Description Techniques (FORTE'88)*, pages 107–119. North-Holland, 1989.

[78] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986 (part II)*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag, 1987.

[79] A. Meyer and S. Cosmadakis. Semantical paradigms: Notes for an invited lecture. In *Proceedings 3rd IEEE Symposium on Logic in Comp. Science*, pages 236–253, 1988.

[80] B. Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[81] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[82] C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.

[83] C. Morgan and B. Sufrin. Specification of the Unix filing system. *IEEE Trans. Software Engineering*, SE-10(2):128–142, 1984. (An updated version appears in [56]).

[84] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[85] K. Narfelt. SYSDAX - an object oriented design methodology based on SDL. In *SDL'87: State of the Art and Future Trends*. North-Holland, 1987.

[86] C. Nix and B. Collins. The use of software engineering, including the Z notation in the development of CICS. IBM Technical Report TR12.266, IBM United Kingdom Laboratories Ltd., Hursley Park, April 1989.

[87] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–6, 1986.

[88] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[89] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall International, 1981.

[90] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[91] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In W. Brauer, editor, *Proceedings 12th International Colloquium on Automata, Languages and Programming (ICALP'85)*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1985.

[92] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.

[93] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Series in Computer Science. Prentice-Hall International, 1990.

[94] U. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[95] A. Roscoe and G. Barrett. Unbounded nondeterminism in CSP. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings 5th International Conference on the Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 160–193. Springer-Verlag, 1990.

[96] S. Rudkin. Inheritance in LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV (FORTE'91)*, pages 415–430, 1991.

[97] J. Russell. Full abstraction for nondeterministic dataflow networks. Technical Report TR 89-1022, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[98] S. Schuman and D. Pitt. Object-oriented subsystem specification. In L. Meertens, editor, *Program Specification and Transformation*, pages 313–341. North-Holland, 1987.

[99] S. Schuman, D. Pitt, and P. Byers. Object-oriented process specification. Technical report, University of Surrey, 1989.

[100] A. Sernadas and C. Sernadas. Object-oriented specification of databases: An algebraic approach. In *Proceedings 13th ACM Conference on Very Large Databases*, 1987.

[101] R. Sijelmassi and P. Gaudette. An object-oriented model for Estelle. In K. Turner, editor, *Formal Description Techniques (FORTE'88)*, pages 91–105. North-Holland, 1989.

[102] L. Simon and L. Marshall. Using VDM to specify OSI managed objects. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV (FORTE'91)*, pages 15–29, 1991.

[103] G. Smith and R. Duke. Modelling a cache coherence protocol using Object-Z. In *Proceedings 13th Australian Computer Science Conference (ACSC-13)*, pages 352–361, 1990.

[104] G. Smith and R. Duke. Specifying concurrent systems using Object-Z. In *Proceedings 15th Australian Computer Science Conference (ACSC-15)*, pages 859–871, January 1992.

[105] A. Snyder. Inheritance and development of encapsulated software components. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 165–188. Springer-Verlag, 1987.

[106] I. Holm Sørensen. A specification language. In J. Staunstrup, editor, *Program Specification*, volume 134 of *Lecture Notes in Computer Science*, pages 381–401. Springer-Verlag, 1982.

[107] J.M. Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, 1988.

[108] J.M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989.

[109] Standards Association of Australia. *Software Quality Management System*, 1991. AS 3563-1991.

[110] S. Stepney, R. Barden, and D. Cooper. Comparative study of object orientation in Z. Technical Report ASE TR 1, Advanced Software Engineering Division, Logica Cambridge Limited, February 1991.

[111] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoret. Computer Science. Pitman/Wiley, 1988.

[112] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[113] UK Ministry of Defence. *The Procurement of Safety Critical Software in Defence Equipment*, 1991. Interim Defence Standard 00-55/Issue 1.

[114] M. Utting and K. Robinson. Towards an object-oriented refinement calculus. In *Proceedings 14th Australian Computer Science Conference (ACSC-14)*, February 1991.

[115] Y. Wand. A proposal for a formal model of objects. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 21, pages 537–559. ACM Press, 1989.

[116] P. Wegner. Dimensions of object-based language design. In *Proceedings 2nd ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, pages 168–182, 1987.

[117] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer-Verlag, 1988.

[118] P. Whysall and J. McDermid. An approach to object oriented specification using Z. In J.E. Nicholls, editor, *Z User Workshop Oxford 1990*. Springer-Verlag, 1990.

[119] P. Whysall and J. McDermid. Object oriented specification and refinement. In J. Morris and R. Shaw, editors, *4th Refinement Workshop*, pages 150–184. Springer-Verlag, 1991.

[120] A. Wills. Capsules and types in Fresco. In P. America, editor, *Proceedings European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, 1991.

[121] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.

[122] M. Wolczko. Semantics of Smalltalk-80. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1987.

[123] P. Yelland. First steps towards fully abstract semantics for object-oriented languages. In S. Cook, editor, *Proceedings European Conference on Object-Oriented Programming (ECOOP'89)*, pages 347–364. Cambridge University Press, 1989.

Appendix A

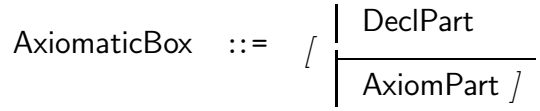
Concrete Syntax of Object-Z

The following concrete syntax of Object-Z is an extension of the concrete syntax of Z presented by Spivey[108]. It is given in an extension to Backus-Naur Form (BNF) defined in [108]. Optional phrases are enclosed in slanted square brackets. NL denotes new line.

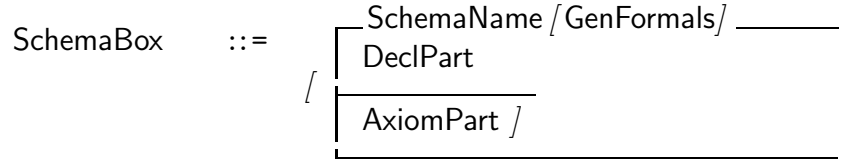
Specification ::= Paragraph NL ... NL Paragraph

Paragraph ::= [Ident, ..., Ident]
| AxiomaticBox
| SchemaBox
| GenericBox
| ClassBox
| SchemaName [GenFormals] $\widehat{=}$ SchemaExp
| ClassName [GenFormals] $\widehat{=}$ ClassRef
| DefLhs == Expression
| Predicate

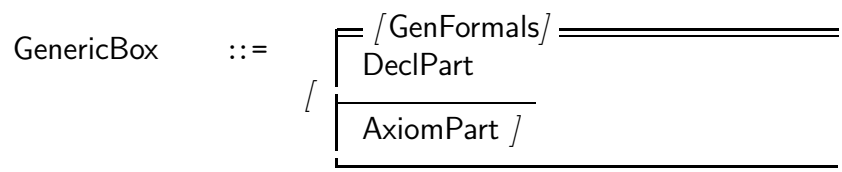
AxiomaticBox ::=



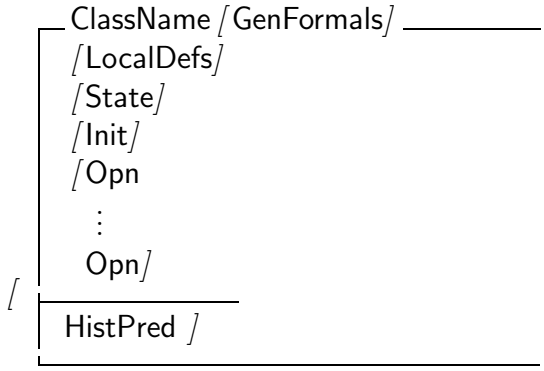
SchemaBox ::=



GenericBox ::=



ClassBox ::=

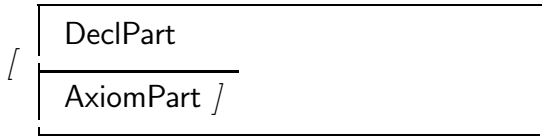


LocalDefs ::= LocalDef
⋮
LocalDef

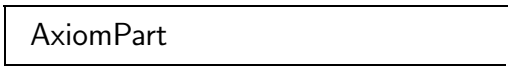
LocalDef ::= InheritedClass
| [Ident, . . . , Ident]
| AxiomaticBox
| DefLhs == Expression

State ::= [DeclPart [| AxiomPart]]
| [AxiomPart]
| StateBox0
| StateBox1

StateBox0 ::=

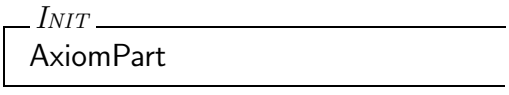


StateBox1 ::=

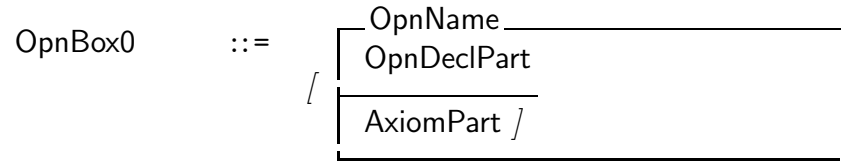


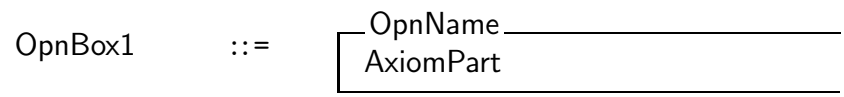
Init ::= $\mathit{INIT} \mathrel{\widehat=} [$ AxiomPart $]$
| InitBox

InitBox ::=



Opn ::= OpnName ≙ OpnExp
| OpnBox0
| OpnBox1

OpnBox0 ::= 

OpnBox1 ::= 

DeclPart ::= BasicDecl Sep ... Sep BasicDecl

OpnDeclPart ::= Δ(DeltaList) [Sep DeclPart]
| DeclPart

AxiomPart ::= Predicate Sep ... Sep Predicate

Sep ::= ; | NL

DefLhs ::= VarName [GenFormals]
| PreGen Ident
| Ident InGen Ident

SchemaExp ::= $\forall$ SchemaText • SchemaExp
| $\exists$ SchemaText • SchemaExp
| $\exists_1$ SchemaText • SchemaExp
| SchemaExp1

SchemaExp1 ::= [SchemaText]
| SchemaRef
| $\neg$ SchemaExp1
| pre SchemaExp1
| SchemaExp1 $\wedge$ SchemaExp1
| SchemaExp1 $\vee$ SchemaExp1
| SchemaExp1 $\Rightarrow$ SchemaExp1
| SchemaExp1 $\Leftrightarrow$ SchemaExp1
| SchemaExp1 $\upharpoonright$ SchemaExp1
| SchemaExp1 $\setminus$ (DeclName, ..., DeclName)
| SchemaExp1 $\fatsemi$ SchemaExp1
| (SchemaExp)

OpnExp ::= $\forall$ SchemaText • OpnExp
| $\exists$ SchemaText • OpnExp
| $\exists_1$ SchemaText • OpnExp
| OpnExp1

OpnExp1 ::= [OpnText]
| OpnRef
| ¬ OpnExp1
| pre OpnExp1
| OpnExp1 ∧ OpnExp1
| OpnExp1 ∨ OpnExp1
| OpnExp1 ∥ OpnExp1
| OpnExp1 ⇒ OpnExp1
| OpnExp1 ⇔ OpnExp1
| OpnExp1 ↾ OpnExp1
| OpnExp1 \ (DeclName, . . . , DeclName)
| OpnExp1 • OpnExp1
| (OpnExp)

SchemaText ::= Declaration [| Predicate]

OpnText ::= OpnDeclaration [| Predicate]
| Predicate

SchemaRef ::= SchemaName Decoration [GenActuals]

InheritedClass ::= ClassRef [RenameList] [RedefList]

ClassRef ::= ClassName [GenActuals]

OpnRef ::= OpnName
| ObjRef.OpnName

Declaration ::= BasicDecl; . . . ; BasicDecl

OpnDeclaration ::= Δ(DeltaList) [; Declaration]
| Declaration

BasicDecl ::= DeclName, . . . , DeclName : Type
| SchemaRef
| OpnRef

Type ::= Expression
| ClassRef
| ↓ClassRef

Predicate ::= $\forall$ SchemaText • Predicate
| $\exists$ SchemaText • Predicate
| $\exists_1$ SchemaText • Predicate
| Predicate1

Predicate1 ::= Expression Rel Expression Rel . . . Rel Expression
| PreRel Expression
| SchemaRef
| OpnRef
| ObjRef.INIT
| pre SchemaRef
| pre OpnRef
| true
| false
| $\neg$ Predicate1
| Predicate1 $\wedge$ Predicate1
| Predicate1 $\vee$ Predicate1
| Predicate1 $\Rightarrow$ Predicate1
| Predicate1 $\Leftrightarrow$ Predicate1
| (Predicate)

Rel ::= $=$ | $\in$ | InRel

Expression0 ::= λ SchemaText • Expression
| Expression

Expression ::= Expression InRel Expression
| Expression1 $\times$ Expression1 $\times$ . . . $\times$ Expression1
| Expression1

Expression1 ::= Expression1 InFun Expression1
| $\mathbb{P}$ Expression3
| PreGen Expression3
| $-$Expression3
| Expression3 PostFun
| Expression3$^{\text{Expression}}$
| Expression3(| Expression0 |)
| Expression2

Expression2 ::= Expression2 Expression3
| Expression3

Expression3	::=	VarName $[$GenActuals$]$
	|	Number
	|	SchemaRef
	|	SetExp
	|	$\langle [$Expression, ..., Expression$] \rangle$
	|	$[\![[$Expression, ..., Expression$]]\!]$
	|	(Expression, ..., Expression)
	|	θSchemaName Decoration
	|	Expression3.VarName
	|	(Expression0)
HistPred	::=	$\forall$ Declaration $[$ | HistPred$]$ • HistPred
	|	$\exists$ Declaration $[$ | HistPred$]$ • HistPred
	|	$\exists_1$ Declaration $[$ | HistPred$]$ • HistPred
	|	HistPred1
HistPred1	::=	Predicate
	|	$\overrightarrow{\text{ObjRef}}$
	|	OpnRef **enabled** $[$ | Predicate$]$
	|	OpnRef **occurs** $[$ | Predicate$]$
	|	$\Box$HistPred1
	|	$\Diamond$HistPred1
	|	$\neg$ HistPred1
	|	HistPred1 $\wedge$ HistPred1
	|	HistPred1 $\vee$ HistPred1
	|	HistPred1 $\Rightarrow$ HistPred1
	|	HistPred1 $\Leftrightarrow$ HistPred1
	|	(HistPred1)
RenameList	::=	[RenItem, ..., RenItem]
RenItem	::=	FeatureRen | ParamRen
FeatureRen	::=	Ident/Ident
ParamRen	::=	OpnName[FeatureRen, ..., FeatureRen]
RedefList	::=	[**redef** OpnName, ..., OpnName]
DeltaList	::=	Ident, ..., Ident

SetExp	::=	$\{ [\mathsf{Expression}, \ldots, \mathsf{Expression}] \}$
	|	$\{\mathsf{SchemaText} \mid \bullet \mathsf{Expression}\}$
ObjRef	::=	Ident | (Ident, Ident)
Ident	::=	Word Decoration
DeclName	::=	Ident | OpName
VarName	::=	Ident | (OpName)
OpName	::=	_InSym_ | PreSym_ | _PostSym | _(| _ |) | $-$
InSym	::=	InFun | InGen | InRel
PreSym	::=	PreGen | PreRel
PostSym	::=	PostFun
Decoration	::=	$[\mathsf{Stroke} \ldots \mathsf{Stroke}]$
GenFormals	::=	[Ident, . . . , Ident]
GenActuals	::=	[Expression, . . . , Expression]

Word	Undecorated name or special symbol
Stroke	Single decoration: ′, ?, ! or a subscript digit
SchemaName	Same as Word, but used to name a schema
OpnName	Same as Word, but used to name an operation
ClassName	Same as Word, but used to name a class
InFun	Infix function symbol $\mapsto \ .. \ + \ - \ \cup \ \setminus \ \frown \ * \ \mathsf{div} \ \mathsf{mod} \ \cap \ \upharpoonright \ \fatsemi \ \circ \ \oplus \ \lhd \ \rhd \ \lhd \ \rhd$
InRel	Infix relation symbol $\neq \notin \subseteq \subset < \leqslant \geqslant > \ \text{partitions}$
InGen	Infix generic symbol $\leftrightarrow \ \pfun \ \fun \ \rightarrowtail \ \fun \ \surj \ \fun \ \rightarrowtail \ \surj$
PreRel	Prefix relation symbol disjoint
PreGen	Prefix generic symbol $\mathbb{P}_1 \ \text{id} \ \mathbb{F} \ \mathbb{F}_1 \ \text{seq} \ \text{seq}_1 \ \text{seq}_\infty \ \text{bag}$
PostFun	Postfix function symbol $\_^\sim \ \_^* \ \_^+$
Number	Unsigned decimal integer

Appendix A. Concrete Syntax of Object-Z

Appendix B

Glossary of Z Notation

This appendix presents a glossary of the Z notation used in this thesis. The glossary is based on the glossary of Z notation presented in Hayes[56] with modifications to reflect more closely the more recent Z notation of Spivey[108].

Mathematical Notation

Definitions and declarations

Let x, x_k be identifiers and let T, T_k be non-empty, set-valued expressions.

$LHS == RHS$	Definition of LHS as syntactically equivalent to RHS.
$LHS[X_1, X_2, \ldots, X_n] == RHS$	Generic definition of LHS, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets.
$x : T$	A declaration, $x : T$, introduces a new variable x of type T.
$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$	List of declarations.
$x_1, x_2, \ldots, x_n : T$	$== x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$
$[X_1, X_2, \ldots, X_n]$	Introduction of free types named $X_1, X_2, \ldots, X_n$.

Logic

Let P, Q be predicates and let D be a declaration or a list of declarations.

$true, false$	Logical constants.
$\neg\, P$	Negation: "not P".
$P \land Q$	Conjunction: "P and Q".
$P \lor Q$	Disjunction: "P or Q or both".
$P \Rightarrow Q$	$== (\neg\, P) \lor Q$ Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	$== (P \Rightarrow Q) \land (Q \Rightarrow P)$ Equivalence: "P is logically equivalent to Q".
$\forall\, x : T \bullet P$	Universal quantification: "for all x of type T, P holds".
$\exists\, x : T \bullet P$	Existential quantification: "there exists an x of type T such that P holds".
$\exists_1\, x : T \bullet P$	Unique existence: "there exists a unique x of type T such that P holds".
$\forall\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	"For all x_1 of type T_1, x_2 of type T_2, ..., and x_n of type T_n, P holds."
$\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\exists_1\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\forall\, D \mid P \bullet Q$	$\Leftrightarrow \forall\, D \bullet P \Rightarrow Q$
$\exists\, D \mid P \bullet Q$	$\Leftrightarrow \exists\, D \bullet P \land Q$
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\Leftrightarrow \neg\, (t_1 = t_2)$

Sets

Let X be a set; S and T be subsets of X; t, t_k terms; P a predicate; and D declarations.

$t \in S$	Set membership: "t is a member of S".
$t \notin S$	$\Leftrightarrow \neg\, (t \in S)$
$S \subseteq T$	$\Leftrightarrow (\forall\, x : S \bullet x \in T)$ Set inclusion.
$S \subset T$	$\Leftrightarrow S \subseteq T \land S \neq T$ Strict set inclusion.
$\varnothing$	The empty set.
$\{t_1, t_2, \ldots, t_n\}$	The set containing the values of terms $t_1, t_2, \ldots, t_n$.
$\{x : T \mid P\}$	The set containing exactly those x of type T for which P holds.
$(t_1, t_2, \ldots, t_n)$	Ordered n-tuple of $t_1, t_2, \ldots, t_n$.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the kth component is of type T_k.
$\mathit{first}(t_1, t_2, \ldots, t_n)$	$== t_1$ Similarly, $\mathit{second}(t_1, t_2, \ldots, t_n) == t_2$, etc.
$\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$	The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.
$\{D \mid P \bullet t\}$	The set of values of the term t for the variables declared in D ranging over all values for which P holds.
$\{D \bullet t\}$	$== \{D \mid \mathit{true} \bullet t\}$
$\mathbb{P}\, S$	Powerset: the set of all subsets of S.
$\mathbb{P}_1\, S$	$== \mathbb{P}\, S \setminus \{\varnothing\}$ The set of all non-empty subsets of S.
$\mathbb{F}\, S$	$== \{\, T : \mathbb{P}\, S \mid T \text{ is finite } \}$ Set of finite subsets of S.

$\mathbb{F}_1\, S$ $== \mathbb{F}\, S \setminus \{\varnothing\}$
Set of finite non-empty subsets of S.

$S \cap T$ $== \{x : X \mid x \in S \land x \in T\}$
Set intersection.

$S \cup T$ $== \{x : X \mid x \in S \lor x \in T\}$
Set union.

$S \setminus T$ $== \{x : X \mid x \in S \land x \notin T\}$
Set difference.

$\bigcap SS$ $== \{x : X \mid (\forall\, S : SS \bullet x \in S)\}$
Intersection of a set of sets; SS is a set containing as its members subsets of X, i.e. $SS : \mathbb{P}(\mathbb{P}\, X)$.

$\bigcup SS$ $== \{x : X \mid (\exists\, S : SS \bullet x \in S)\}$
Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\, X)$.

$\#S$ Size (number of distinct members) of a finite set.

Numbers

$\mathbb{R}$ The set of real numbers.

$\mathbb{Z}$ The set of integers (positive, zero and negative).

$\mathbb{N}$ $== \{n : \mathbb{Z} \mid n \geq 0\}$
The set of natural numbers (non-negative integers).

$\mathbb{N}_1$ $== \mathbb{N} \setminus \{0\}$
The set of strictly positive natural numbers.

$m \mathinner{..} n$ $== \{k : \mathbb{Z} \mid m \leq k \land k \leq n\}$
The set of integers between m and n inclusive.

$min\, S$ Minimum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$,
$min\, S \in S \land (\forall\, x : S \bullet x \geq min\, S)$.

$max\, S$ Maximum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$,
$max\, S \in S \land (\forall\, x : S \bullet x \leq max\, S)$.

Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$ $== \mathbb{P}(X \times Y)$
The set of relations between X and Y.

$x \mathrel{\underline{R}} y$ $== (x, y) \in R$
x is related by R to y.

$x \mapsto y$ $== (x, y)$

$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$
$== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
The relation relating x_1 to y_1, x_2 to y_2, ..., and x_n to y_n.

$\operatorname{dom} R$ $== \{x : X \mid (\exists\, y : Y \bullet x \mathrel{\underline{R}} y)\}$
The domain of a relation: the set of x components that are related to some y.

$\operatorname{ran} R$ $== \{y : Y \mid (\exists\, x : X \bullet x \mathrel{\underline{R}} y)\}$
The range of a relation: the set of y components that some x is related to.

$R_1 \fatsemi R_2$ $== \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \land y\, R_2\, z)\}$
Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$.

$R_1 \circ R_2$ $== R_2 \fatsemi R_1$
Relational composition. This form is primarily used when R_1 and R_2 are functions.

$R^{\sim}$ $== \{y : Y;\ x : X \mid x \mathrel{\underline{R}} y\}$
Transpose of a relation R.

$\operatorname{id} S$ $== \{x : S \bullet x \mapsto x\}$
Identity function on the set S.

R^k The homogeneous relation R composed with itself k times: given $R : X \leftrightarrow X$,
$R^0 = \operatorname{id} X$ and $R^{k+1} = R^k \fatsemi R$.

R^+ $== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$
$= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
Transitive closure.

R^* $== \bigcup\{n : \mathbb{N} \bullet R^n\}$
$= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
Reflexive transitive closure.

$R(\!| S |\!)$ $== \{y : Y \mid (\exists\, x : S \bullet x \mathrel{\underline{R}} y)\}$
Image of the set S through the relation R.

$S \lhd R$ $== \{x : X;\ y : Y \mid x \in S \land x \mathrel{\underline{R}} y\}$
Domain restriction: the relation R with its domain restricted to the set S.

$S \lhd\!\!\!- R$ $== (X \setminus S) \lhd R$
Domain subtraction: the relation R with the elements of S removed from its domain.

$R \rhd T$ $== \{x : X;\ y : Y \mid x \mathrel{\underline{R}} y \land y \in T\}$
Range restriction to T.

$R \rhd\!\!\!- T$ $== R \rhd (Y \setminus T)$
Range subtraction of T.

$R_1 \oplus R_2$ $== (\mathrm{dom}\, R_2 \lhd\!\!\!- R_1) \cup R_2$
Overriding; $R_1, R_2 : X \leftrightarrow Y$.

Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$ The function f applied to t.

$X \pfun Y$ $== \{f : X \leftrightarrow Y \mid (\forall\, x : \mathrm{dom}\, f \bullet (\exists_1\, y : Y \bullet x \mathrel{\underline{f}} y))\}$
The set of partial functions from X to Y.

$X \fun Y$ $== \{f : X \pfun Y \mid \mathrm{dom}\, f = X\}$
The set of total functions from X to Y.

$X \pinj Y$ $== \{f : X \pfun Y \mid (\forall\, y : \mathrm{ran}\, f \bullet (\exists_1\, x : X \bullet x \mathrel{\underline{f}} y))\}$
The set of partial one-to-one functions (partial injections) from X to Y.

$X \rightarrowtail Y$ $== \{f : X \pfun Y \mid \dom f = X\}$
The set of total one-to-one functions (total injections) from X to Y.

$X \twoheadrightarrow Y$ $== \{f : X \pfun Y \mid \ran f = Y\}$
The set of partial onto functions (partial surjections) from X to Y.

$X \rightarrow\!\!\!\!\!\twoheadrightarrow Y$ $== (X \twoheadrightarrow Y) \cap (X \fun Y)$
The set of total onto functions (total surjections) from X to Y.

$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$ $== (X \rightarrow\!\!\!\!\!\twoheadrightarrow Y) \cap (X \rightarrowtail Y)$
The set of total one-to-one onto functions (total bijections) from X to Y.

$X \finfun Y$ $== \{f : X \pfun Y \mid f \in \finset(X \times Y)\}$
The set of finite partial functions from X to Y.

$X \finfun\!\!\!\!\!\rightarrowtail Y$ $== \{f : X \rightarrowtail Y \mid f \in \finset(X \times Y)\}$
The set of finite partial one-to-one functions from X to Y.

$(\lambda\, x : X \mid P \bullet t)$ $== \{x : X \mid P \bullet x \mapsto t\}$
Lambda-abstraction: the function that, given an argument x of type X such that P holds, gives a result which is the value of the term t.

$(\lambda\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet t)$
$== \{x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$

$\mathrm{disjoint}[I, X]$ $== \{S : I \pfun \power X \mid \forall\, i, j : \dom S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \varnothing\}$
Pairwise disjoint; where I is a set and S an indexed family of subsets of X (i.e. $S : I \pfun \power X$).

S partitions T $== S \in \mathrm{disjoint} \land \bigcup \ran S = T$

Sequences

Let X be a set; A and B be sequences with elements taken from X; and $a_1, \ldots, a_n$ terms of type X.

$\seq X$ $== \{A : \natone \pfun X \mid (\exists\, n : \nat \bullet \dom A = 1..n)\}$
The set of finite sequences whose elements are drawn from X.

$\text{seq}_\infty X$ $== \{A : \mathbb{N}_1 \pfun X \mid A \in \text{seq}\, X \lor \text{dom}\, A = \mathbb{N}_1\}$
The set of finite and infinite sequences whose elements are drawn from X.

$\#A$ The length of a finite sequence A. (This is just '#' on the set representing the sequence.)

$\langle\rangle$ $== \{\}$
The empty sequence.

$\text{seq}_1 X$ $== \{s : \text{seq}\, X \mid s \neq \langle\rangle\}$
The set of non-empty finite sequences.

$\langle a_1, \ldots, a_n\rangle$ $= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$

$\langle a_1, \ldots, a_n\rangle \frown \langle b_1, \ldots, b_m\rangle$
$= \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$
Concatenation.
$\langle\rangle \frown A = A \frown \langle\rangle = A$.

head A The first element of a non-empty sequence:
$A \neq \langle\rangle \Rightarrow$ *head* $A = A(1)$.

tail A All but the head of a non-empty sequence:
tail $(\langle x\rangle \frown A) = A$.

last A The final element of a non-empty finite sequence:
$A \neq \langle\rangle \Rightarrow$ *last* $A = A(\#A)$.

front A All but the last of a non-empty finite sequence:
front $(A \frown \langle x\rangle) = A$.

rev $\langle a_1, a_2, \ldots, a_n\rangle$
$= \langle a_n, \ldots, a_2, a_1\rangle$
Reverse of a finite sequence; *rev* $\langle\rangle = \langle\rangle$.

$\frown / AA$ $= AA(1) \frown \ldots \frown AA(\#AA)$
Distributed concatenation; where $AA : \text{seq}(\text{seq}(X))$. $\frown/\langle\rangle = \langle\rangle$.

$A \subseteq B$ $\Leftrightarrow \exists\, C : \text{seq}_\infty X \bullet A \frown C = B$
A is a prefix of B. (This is just '$\subseteq$' on the sets representing the sequences.)

squash f Convert a finite function, $f : \mathbb{N} \pfun X$, into a sequence by squashing its domain. That is, $squash\{\} = \langle\rangle$, and if $f \neq \{\}$ then $squash\, f = \langle f(i)\rangle \frown squash(\{i\} \ndres f)$, where $i = min(\text{dom}\, f)$. For example, $squash\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C\rangle$.

$A \upharpoonright T$ $== squash(A \rhd T)$
Restrict the range of the sequence A to the set T.

Bags

$\mathrm{bag}\, X$	$== X \pfun \mathbb{N}_1$ The set of bags whose elements are drawn from X. A bag is represented by a function that maps each element in the bag onto its frequency of occurrence in the bag.
$[\![\,]\!]$	The empty bag $\varnothing$.
$[\![x_1, x_2, \ldots, x_n]\!]$	The bag containing $x_1, x_2, \ldots, x_n$, each with the frequency that it occurs in the list.
$items\ s$	$== \{x : \mathrm{ran}\, s \bullet x \mapsto \#\{i : \mathrm{dom}\, s \mid s(i) = x\}\}$ The bag of items contained in the sequence s.

Axiomatic definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D. These variables must satisfy the predicate P. The scope of the variables is the whole specification.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

Generic definitions

Let D be a list of declarations, P a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$\begin{array}{|l} [X_1, X_2, \ldots X_n] \\ D \\ \hline P \end{array}$$

The declared variables must be uniquely defined by the predicate P.

Schema Notation

Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$
\begin{array}{|l}
\hline
S \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \\
\hline
\end{array}
$$

and horizontally, for the same example,

$$S == [x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y]$$

Schemas can be used in signatures after $\forall$, λ, $\{...\}$, etc.:

$$(\forall S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y \bullet y \neq \langle\rangle)$$

$\{S\}$	Stands for the set of objects described by schema S. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$.

Schema operators

Let S be defined as above and $w : S$.

$w.x$	$== (\lambda S \bullet x)(w)$ Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \# w.y$' holds.
θS	The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y.
Compatibility	Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must have the same declared sets) and the predicates of S and T are conjoined. For example,

$$\begin{array}{|l} T \\ \hline S \\ z : \mathbb{N} \\ \hline z < x \\ \hline \end{array}$$

is equivalent to

$$\begin{array}{|l} T \\ \hline x, z : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ \hline x \leq \# y \land z < x \\ \hline \end{array}$$

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (T).

Decoration Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example, S' is
$[x' : \mathbb{N};\ y' : \operatorname{seq} \mathbb{N} \mid x' \leq \# y']$.

$\neg\, S$ The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid \neg\, (x \leq \# y)]$.

$S \land T$ The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above).
Given $T == [x : \mathbb{N};\ z : \mathbb{P}\, \mathbb{N} \mid x \in z]$, $S \land T$ is

$$\begin{array}{|l} S \land T \\ \hline x : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ z : \mathbb{P}\, \mathbb{N} \\ \hline x \leq \# y \land x \in z \\ \hline \end{array}$$

$S \lor T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \lor T$ is

$S \vee T$

$$x : \mathbb{N}$$
$$y : \operatorname{seq} \mathbb{N}$$
$$z : \mathbb{P}\, \mathbb{N}$$
$$x \leq \# y \vee x \in z$$

$S \Rightarrow T$

The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Rightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$S \Rightarrow T$

$$x : \mathbb{N}$$
$$y : \operatorname{seq} \mathbb{N}$$
$$z : \mathbb{P}\, \mathbb{N}$$
$$x \leq \# y \Rightarrow x \in z$$

$S \Leftrightarrow T$

The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Leftrightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$S \Leftrightarrow T$

$$x : \mathbb{N}$$
$$y : \operatorname{seq} \mathbb{N}$$
$$z : \mathbb{P}\, \mathbb{N}$$
$$x \leq \# y \Leftrightarrow x \in z$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The parantheses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

$(S \wedge T) \upharpoonright (x, y)$

$$x : \mathbb{N}$$
$$y : \operatorname{seq} \mathbb{N}$$
$$(\exists\, z : \mathbb{P}\, \mathbb{N} \bullet x \leq \# y \wedge x \in z)$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists D \bullet S$ Existential quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is existentially quantified over D. For example, $\exists x : \mathbb{N} \bullet S$ is the following schema.

$\exists x : \mathbb{N} \bullet S$

$y : \operatorname{seq} \mathbb{N}$

$\exists x : \mathbb{N} \bullet$
$\quad x \leq \# y$

The declarations may include schemas. For example,

$\exists S \bullet T$

$z : \mathbb{N}$

$\exists S \bullet$
$\quad x \leq \# y \land z < x$

$\forall D \bullet S$ Universal quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is universally quantified over D. For example, $\forall x : \mathbb{N} \bullet S$ is the following schema.

$\forall x : \mathbb{N} \bullet S$

$y : \operatorname{seq} \mathbb{N}$

$\forall x : \mathbb{N} \bullet$
$\quad x \leq \# y$

The declarations may include schemas. For example,

$\forall S \bullet T$

$z : \mathbb{N}$

$\forall S \bullet$
$\quad x \leq \# y \land z < x$

Operation schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

undashed state before the operation,

dashed state after the operation,

ending in "?" inputs to (arguments for) the operation, and

ending in "!" outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS $\quad \widehat{=} S \land S'$
Change of state schema: this is a default definition for ΔS. In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\quad \widehat{=} [\Delta S \mid \theta S' = \theta S]$
No change of state schema.

Operation schema operators

pre S Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

S

$$x?, s, s', y! : \mathbb{N}$$

$$s' = s - x? \land y! = s'$$

pre S is,

pre S

$$x?, s : \mathbb{N}$$

$$\exists\, s', y! : \mathbb{N} \bullet s' = s - x? \land y! = s'$$

$S \fatsemi T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S \fatsemi T$ is,

$S \fatsemi T$

$x?, s, s', y! : \mathbb{N}$

$(\exists\, ss : \mathbb{N} \bullet$
$\quad ss = s - x? \land y! = ss$
$\quad \land\ ss \leq x? \land s' = ss + x?)$

Appendix C

Proof of Lemmas

This appendix includes the proofs of the lemmas required for the proof of compositionality of the complete-readiness model in Chapter 5. A denotes a class and A its structural model. C denotes a context which includes a single object a of its elided class as a state variable.

Lemma C.1

The history of an object of class A is in the set $h\_init(\mathsf{A})$ if and only if its behaviour is in $b\_init(\mathsf{A})$.

$$behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = b\_init(\mathsf{A})$$

Proof

From the definition of $behav(\mathsf{A})$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, h : h\_init(\mathsf{A}) \bullet \\ b.events = h.events \land \\ \forall\, i : b.readys;\ ph : prehist(h) \bullet \\ \#ph.states = i \Rightarrow \\ b.readys(i) = next(\mathsf{A}, ph)\} \end{aligned}$$

From the definition of $h\_init(\mathsf{A})$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ h.events = \langle\,\rangle \land \\ b.events = h.events \land \\ \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\ \#ph.states = i \Rightarrow \\ b.readys(i) = next(\mathsf{A}, ph)\} \end{aligned}$$

Since $h.events = \langle\,\rangle \wedge b.events = h.events$ is equivalent to $b.events = \langle\,\rangle \wedge b.events = h.events$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events = \langle\,\rangle \wedge \\ & \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad b.events = h.events \wedge \\ & \quad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\ & \quad\quad \#ph.states = i \Rightarrow \\ & \quad\quad\quad b.readys(i) = next(\mathsf{A}, ph)\} \end{aligned}$$

From the definition of $behav(\mathsf{A})$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events = \langle\,\rangle \wedge \\ & \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet b = behav(\mathsf{A})(h)\} \end{aligned}$$

Since $\mathcal{CR}_{safe}(\mathsf{A}) = behav(\mathsf{A})(\!|\ \mathcal{TH}_{safe}(\mathsf{A})\ |\!)$,

$$behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid b.events = \langle\,\rangle\}$$

From the definition of $b\_init(\mathsf{A})$,

$$behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = b\_init(\mathsf{A})$$

□

Lemma C.2

The history of an object of class A is in $h\_pre(\mathsf{A}, e)$, for a particular event e, if and only if its behaviour is in $b\_pre(\mathsf{A}, e)$.

$$behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = b\_pre(\mathsf{A}, e)$$

Proof

From the definition of $behav(\mathsf{A})$,

$$\begin{aligned}
&behav(\mathsf{A})(\!|\ h_pre(\mathsf{A}, e)\ |\!) = \\
&\quad \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, h : h_pre(\mathsf{A}, e) \bullet \\
&\qquad\qquad b.events = h.events \land \\
&\qquad\qquad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\
&\qquad\qquad\quad \#ph.states = i \Rightarrow \\
&\qquad\qquad\qquad b.readys(i) = next(\mathsf{A}, ph)\}
\end{aligned}$$

From the definition of $h\_pre(\mathsf{A}, e)$,

$$\begin{aligned}
&behav(\mathsf{A})(\!|\ h_pre(\mathsf{A}, e)\ |\!) = \\
&\quad \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
&\qquad\qquad b.events = h.events \land \\
&\qquad\qquad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\
&\qquad\qquad\quad \#ph.states = i \Rightarrow \\
&\qquad\qquad\qquad b.readys(i) = next(\mathsf{A}, ph) \land \\
&\qquad\qquad h.events \in \mathrm{seq}\, Event \land \\
&\qquad\qquad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
&\qquad\qquad\quad front\, h'.states = h.states \land \\
&\qquad\qquad\quad h'.events = h.events \frown \langle e \rangle\}
\end{aligned}$$

Since $\mathcal{CR}_{safe}(\mathsf{A}) = behav(\mathsf{A})(\!|\ \mathcal{TH}_{safe}(\mathsf{A})\ |\!)$,

$$\begin{aligned}
&behav(\mathsf{A})(\!|\ h_pre(\mathsf{A}, e)\ |\!) = \\
&\quad \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\
&\qquad\qquad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
&\qquad\qquad\quad b.events = h.events \land \\
&\qquad\qquad\quad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\
&\qquad\qquad\qquad \#ph.states = i \Rightarrow \\
&\qquad\qquad\qquad\quad b.readys(i) = next(\mathsf{A}, ph) \land \\
&\qquad\qquad\quad h.events \in \mathrm{seq}\, Event \land \\
&\qquad\qquad\quad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
&\qquad\qquad\qquad behav(\mathsf{A})(h') = b' \\
&\qquad\qquad\qquad front\, h'.states = h.states \land \\
&\qquad\qquad\qquad h'.events = h.events \frown \langle e \rangle\}
\end{aligned}$$

From the definition of $behav(\mathsf{A})$,

$$
\begin{array}{l}
behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \\
\quad \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\
\qquad\qquad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
\qquad\qquad\quad b.events = h.events \land \\
\qquad\qquad\quad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\
\qquad\qquad\qquad \#ph.states = i \Rightarrow \\
\qquad\qquad\qquad\quad b.readys(i) = next(\mathsf{A}, ph) \land \\
\qquad\qquad\quad h.events \in \mathrm{seq}\, Event \land \\
\qquad\qquad\quad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
\qquad\qquad\qquad b'.events = h'.events \land \\
\qquad\qquad\qquad \forall\, i : \mathrm{dom}\, b'.readys;\ ph : prehist(h') \bullet \\
\qquad\qquad\qquad\quad \#ph.states = i \Rightarrow \\
\qquad\qquad\qquad\qquad b'.readys(i) = next(\mathsf{A}, ph) \land \\
\qquad\qquad\qquad front\, h'.states = h.states \land \\
\qquad\qquad\qquad h'.events = h.events \frown \langle e \rangle\}
\end{array}
$$

Since $b'.events = h'.events \land b.events = h.events \land h'.events = h.events \frown \langle e \rangle$ is equivalent to $b'.events = h'.events \land b.events = h.events \land b'.events = b.events \frown \langle e \rangle$, and $b.events = h.events \land h.events \in \mathrm{seq}\, Event$ is equivalent to $b.events = h.events \land b.events \in \mathrm{seq}\, Event$,

$$
\begin{array}{l}
behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \\
\quad \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid b.events \in \mathrm{seq}\, Event \\
\qquad\qquad \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\
\qquad\qquad\quad b'.events = b.events \frown \langle e \rangle \land \\
\qquad\qquad\quad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\
\qquad\qquad\qquad b.events = h.events \land \\
\qquad\qquad\qquad \forall\, i : \mathrm{dom}\, b.readys;\ ph : prehist(h) \bullet \\
\qquad\qquad\qquad\quad \#ph.states = i \Rightarrow \\
\qquad\qquad\qquad\qquad b.readys(i) = next(\mathsf{A}, ph) \land \\
\qquad\qquad\qquad \exists\, h' : \mathcal{TH}_{safe}(A) \bullet \\
\qquad\qquad\qquad\quad b'.events = h'.events \land \\
\qquad\qquad\qquad\quad \forall\, i : \mathrm{dom}\, b'.readys;\ ph : prehist(h') \bullet \\
\qquad\qquad\qquad\qquad \#ph.states = i \Rightarrow \\
\qquad\qquad\qquad\qquad\quad b'.readys(i) = next(\mathsf{A}, ph) \land \\
\qquad\qquad\qquad\quad front\, h'.states = h.states\}
\end{array}
$$

From the definition of $behav(\mathsf{A})$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events \in \mathrm{seq}\, Event \\ & \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\ & \quad b'.events = b.events \frown \langle e \rangle \wedge \\ & \quad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad behav(\mathsf{A})(h) = b \wedge \\ & \quad\quad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad\quad behav(\mathsf{A})(h') = b' \wedge \\ & \quad\quad\quad front\ h'.states = h.states\} \end{aligned}$$

Since $\exists\, h, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet front\ h'.states = h.states \wedge behav(\mathsf{A})(h) = b \wedge behav(\mathsf{A})(h') = b'$ implies $front\ b'.readys = b.readys$ (from the definition of $behav(\mathsf{A})$),

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events \in \mathrm{seq}\, Event \\ & \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\ & \quad front\ b'.readys = b.readys \wedge \\ & \quad b'.events = b.events \frown \langle e \rangle \wedge \\ & \quad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad behav(\mathsf{A})(h) = b \wedge \\ & \quad\quad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad\quad behav(\mathsf{A})(h') = b' \wedge \\ & \quad\quad\quad front\ h'.states = h.states\} \end{aligned}$$

Since $\mathcal{TH}_{safe}(\mathsf{A})$ includes all pre-histories of any history it contains, for all $h' : \mathcal{TH}_{safe}(\mathsf{A})$, there exists a $h : \mathcal{TH}_{safe}(\mathsf{A})$ such that $front\ h'.states = h.states$.

Hence, $front\ b'.readys = b.readys \wedge \exists\, h, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet front\ h'.states = h.states \wedge behav(\mathsf{A})(h) = b \wedge behav(\mathsf{A})(h') = b'$ is equivalent to $front\ b'.readys = b.readys \wedge \exists\, h, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet behav(\mathsf{A})(h) = b \wedge behav(\mathsf{A})(h') = b'$. Therefore,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events \in \mathrm{seq}\, Event \\ & \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\ & \quad front\ b'.readys = b.readys \wedge \\ & \quad b'.events = b.events \frown \langle e \rangle \wedge \\ & \quad \exists\, h : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad behav(\mathsf{A})(h) = b \wedge \\ & \quad\quad \exists\, h' : \mathcal{TH}_{safe}(\mathsf{A}) \bullet \\ & \quad\quad\quad behav(\mathsf{A})(h') = b'\} \end{aligned}$$

Since $\mathcal{CR}_{safe}(\mathsf{A}) = behav(\mathsf{A})(\!|\ \mathcal{TH}_{safe}(\mathsf{A})\ |\!)$,

$$\begin{aligned} behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = \{b : \mathcal{CR}_{safe}(\mathsf{A}) \mid\ & b.events \in \mathrm{seq}\, Event \\ & \exists\, b' : \mathcal{CR}_{safe}(\mathsf{A}) \bullet \\ & \quad front\ b'.readys = b.readys \\ & \quad b'.events = b.events \frown \langle e \rangle\} \end{aligned}$$

From the definition of $b\_pre(\mathsf{A}, e)$,

$$behav(\mathsf{A})(\!| \ h\_pre(\mathsf{A}, e) \ |\!) = b\_pre(\mathsf{A}, e)$$

□

Lemma C.3

A state s of $C[A]$ satisfies a predicate in $C[A]$ using Method 1 if and only if $s\_map(\mathsf{A})(s)$ satisfies the predicate using Method 2.

Proof

1) The predicate does not refer to a.

If the predicate is true given the assignment of values to the variables in s other than a then, using Method 1, it will be true for all histories of a (i.e. all histories in $\mathcal{TH}_{safe}(\mathsf{A})$). Similarly, using Method 2, it will be true for all behaviours of a (i.e. all behaviours in $\mathcal{CR}_{safe}(\mathsf{A})$). Since $behav(\mathsf{A})(\!|\ \mathcal{TH}_{safe}(\mathsf{A})\ |\!) = \mathcal{CR}_{safe}(\mathsf{A})$, $s\_map(\mathsf{A})(s)$ will satisfy the predicate using Method 2 whenever s satisfies the predicate using Method 1.

If the predicate is false given the assignment of values to the variables in s other than a then, using Method 1, the predicate is not satisfied for any history of a. Similarly, using Method 2, the predicate is not satisfied for any behaviour of a.

2) The predicate is $a.\textsc{Init}$.

Using Method 1, the predicate is true when the history of a is in $h\_init(\mathsf{A})$. Using Method 2, the predicate is true when the behaviour of a is in $b\_init(\mathsf{A})$. Since, by Lemma C.1, $behav(\mathsf{A})(\!|\ h\_init(\mathsf{A})\ |\!) = b\_init(\mathsf{A})$, s will satisfy $a.\textsc{Init}$ using Method 1 if and only if $s\_map(\mathsf{A})(s)$ satisfies $a.\textsc{Init}$ using Method 2.

3) The predicate is pre $a.op$ and the assignment of values to $a.op$'s parameters are such that the occurrence of $a.op$ would correspond to the event e.

Using Method 1, the predicate is true when the history of a is in $h\_pre(\mathsf{A}, e)$. Using Method 2, the predicate is true when the behaviour of a is in $b\_pre(\mathsf{A}, e)$. Since by Lemma C.2, $behav(\mathsf{A})(\!|\ h\_pre(\mathsf{A}, e)\ |\!) = b\_pre(\mathsf{A}, e)$, s will satisfy the predicate using Method 1 if and only if $s\_map(\mathsf{A})(s)$ satisfies the predicate using Method 2.

All other predicates in $C[A]$ can be constructed from the predicates above and, using the above results, can be shown to satisfy the lemma. □

Lemma C.4

If the transition (h, h') is in $h\_trans(\mathsf{A}, e)$, for a particular event e, then the transition $(behav(\mathsf{A})(h), behav(\mathsf{A})(h'))$ is in $b\_trans(\mathsf{A}, e)$.

$$\forall (h, h') : h\_trans(\mathsf{A}, e) \bullet (behav(\mathsf{A})(h), behav(\mathsf{A})(h')) \in b\_trans(\mathsf{A}, e)$$

Proof

If (h, h') is in $h\_trans(\mathsf{A}, e)$ then from the definition of $h\_trans(\mathsf{A}, e)$,

$$\begin{array}{l} h.events \in \mathrm{seq}\, Event \land \\ front\, h'.states = h.states \land \\ h'.events = h.events \frown \langle e \rangle \end{array}$$

From the definition of $behav(\mathsf{A})$,

$$\begin{array}{l} behav(\mathsf{A})(h).events = h.events \land \\ \forall\, i : \mathrm{dom}\, behav(\mathsf{A})(h).readys;\ ph : prehist(h) \bullet \\ \quad \#ph.states = i \Rightarrow \\ \quad\quad behav(\mathsf{A})(h).readys(i) = next(\mathsf{A}, ph) \land \\ behav(\mathsf{A})(h').events = h'.events \land \\ \forall\, i : \mathrm{dom}\, behav(\mathsf{A})(h').readys;\ ph : prehist(h') \bullet \\ \quad \#ph.states = i \Rightarrow \\ \quad\quad behav(\mathsf{A})(h').readys(i) = next(\mathsf{A}, ph) \end{array}$$

Therefore,

$$\begin{array}{l} behav(\mathsf{A})(h).events \in \mathrm{seq}\, Event \land \\ front\, behav(\mathsf{A})(h').readys = behav(\mathsf{A})(h).readys \land \\ behav(\mathsf{A})(h').events = behav(\mathsf{A})(h).events \frown \langle e \rangle \end{array}$$

From the definition of $b\_trans(\mathsf{A}, e)$,

$$(behav(\mathsf{A})(h), behav(\mathsf{A})(h')) \in b\_trans(\mathsf{A}, e)$$

□

Lemma C.5

If the transition (b, b') is in $b\_trans(\mathsf{A}, e)$, for a particular event e, then for all h' such that $behav(\mathsf{A})(h') = b'$, there exists a h such that $behav(\mathsf{A})(h) = b$ and the transition (h, h') is in $h\_trans(\mathsf{A}, e)$.

$$\begin{array}{l}\forall (b, b') : b\_trans(\mathsf{A}, e) \bullet \\ \quad \forall\, h' : behav(\mathsf{A})^{\sim}(\!|\, \{b'\} \,|\!) \bullet \exists\, h : behav(\mathsf{A})^{\sim}(\!|\, \{b\} \,|\!) \bullet (h, h') \in h\_trans(\mathsf{A}, e)\end{array}$$

Proof

If (b, b') is in $b\_trans(\mathsf{A}, e)$ and $behav(\mathsf{A})(h') = b'$ then from the definition of $b\_trans(\mathsf{A}, e)$ and $behav(\mathsf{A})$,

$$\begin{array}{l} b \in \mathrm{seq}\, Event \wedge \\ front\, b'.readys = b.readys \wedge \\ b'.events = b.events \frown \langle e \rangle \wedge \\ b'.events = h'.events \wedge \\ \forall\, i : \mathrm{dom}\, b'.readys;\ ph : prehist(h') \bullet \\ \quad \#ph.states = i \Rightarrow \\ \quad\quad b'.readys(i) = next(\mathsf{A}, ph) \end{array}$$

Let h be a history such that $h.states = front\, h'.states$ and $h.events = front\, h'.events$. Since $h \in prehist(h')$, $h \in \mathcal{TH}_{safe}(\mathsf{A})$ and is therefore in the domain of $behav(\mathsf{A})$.

From the definition of $behav(\mathsf{A})$, $behav(\mathsf{A})(h) = b$. Also, from the definition of $h\_trans(\mathsf{A}, e)$, (h, h') is in $h\_trans(\mathsf{A}, e)$. □

Lemma C.6

(a) If a state tuple (s, s') satisfies an operation in $C[A]$ using Method 1 then the tuple $(s\_map(\mathsf{A})(s), s\_map(\mathsf{A})(s'))$ satisfies the operation using Method 2.

(b) If a state tuple (t, t') satisfies an operation in $C[A]$ using Method 2 then for all states s' in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t'\}\ |\!)$, there exists a state s in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t\}\ |\!)$ such that the tuple (s, s') satisfies the operation using Method 1.

Proof

1) The operation does not change a. It may, however, refer to a, i.e. it may involve $a.\textit{INIT}$ or pre $a.op$.

Using Method 1, (s, s') is a transition of the operation if the history of a in s is the same as the history of a in s' and the precondition and postcondition of the operation are true for the assignment of values to the variables in s and s' respectively. Using Method 2, (t, t') is a transition of the operation if the behaviour of a in t is the same as the behaviour of a in t' and the precondition and postcondition of the operation are true for the assignment of values to the variables in t and t' respectively.

(a) Since s satisfies the precondition of the operation using Method 1, $s\_map(\mathsf{A})(s)$ satisfies the precondition using Method 2 by Lemma C.3. Also, since s' satisfies the postcondition of the operation using Method 1, $s\_map(\mathsf{A})(s')$ satisfies the postcondition using Method 2. From the definition of s_map, if the history of a is the same in s and s' then the behaviour of a is the same in $s\_map(\mathsf{A})(s)$ and $s\_map(\mathsf{A})(s')$. Therefore, the operation satisfies part (a) of the lemma.

(b) Since t satisfies the precondition of the operation using Method 2, all states s in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t\}\ |\!)$ satisfy the precondition using Method 1 by Lemma C.3. Also, since t' satisfies the postcondition of the operation using Method 2, all s' in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t'\}\ |\!)$ satisfy the postcondition using Method 1. From the definition of s_map, if the behaviour of a in t is the same as the behaviour of a in t' then given an s' in $s\_map^{\sim}(\mathsf{A})(\!|\ \{t'\}\ |\!)$, there exists an s in $s\_map^{\sim}(\mathsf{A})(\!|\ \{t\}\ |\!)$ such that the history of a in s is the same as the history of a in s'. Therefore, the operation satisfies part (b) of the lemma.

2) The operation changes a arbitrarily, i.e. the operation includes a in its Δ-list but does not refer to a' in its predicate.

Using Method 1, (s, s') is a transition of the operation if the precondition and postcondition of the operation are true for the assignment of values to the variables in s and s' respectively. Using Method 2, (t, t') is a transition of the operation if the precondition and postcondition of the operation are true for the assignment of values to the variables in t and t' respectively.

(a) Since s satisfies the precondition of the operation using Method 1, $s\_map(\mathsf{A})(s)$ satisfies the precondition using Method 2 by Lemma C.3. Also, since s' satisfies the postcondition of the operation using Method 1, $s\_map(\mathsf{A})(s')$ satisfies the postcondition using Method 2. Therefore, the operation satisfies part (a) of the lemma.

(b) Since t satisfies the precondition of the operation using Method 2, all states s in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t\}\ |\!)$ satisfy the precondition using Method 1 by Lemma C.3. Also, since t' satisfies the postcondition of the operation using Method 2, all s' in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t'\}\ |\!)$

satisfy the postcondition using Method 1. Therefore, the operation satisfies part (b) of the lemma.

3) The operation is $a.op$ and the assignment of values to $a.op$'s parameters are such that the occurrence of the operation corresponds to the event e.

Using Method 1, (s, s') is a transition of the operation when the tuple consisting of the history of a in s and the history of a in s' is in $h\_trans(\mathsf{A}, e)$. Using Method 2, (t, t') is a transition of the operation when the tuple consisting of the behaviour of a in t and the behaviour of a in t' is in $b\_trans(\mathsf{A}, e)$.

(a) Let h denote the history of a in s and h' the history of a in s'. Since (h, h') is in $h\_trans(\mathsf{A}, e)$, $(behav(\mathsf{A})(h), behav(\mathsf{A})(h'))$ is in $b\_trans(\mathsf{A}, e)$ by Lemma C.4. Therefore, $(s\_map(\mathsf{A})(s), s\_map(\mathsf{A})(s'))$ satisfies the operation using Method 2.

(b) Let b denote the behaviour of a in t and b' the behaviour of a in t'. Since (b, b') is in $b\_trans(\mathsf{A}, e)$, for all histories h' such that $behav(\mathsf{A})(h') = b'$, there exists a history h such that $behav(\mathsf{A})(h) = b$ and (h, h') is in $h\_trans(\mathsf{A}, e)$ by Lemma C.5. Therefore, for all states s' in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t'\}\ |\!)$, there exists a state s in $s\_map(\mathsf{A})^{\sim}(\!|\ \{t\}\ |\!)$ such that (s, s') satisfies the operation using Method 1.

All other operations in $C[A]$ can be constructed from the operations above and, using the above results, can be shown to satisfy the lemma. □

Lemma C.7

Let H_1 denote the set of safe histories of $C[A]$ derived using Method 1. Given a finite history h_1 in H_1, for any state s such that $s\_map(\mathsf{A})(s) = s\_map(\mathsf{A})(h_1.states(\#h_1.states))$, there exists a history h_1' in H_1 such that $h\_map(\mathsf{A})(h_1') = h\_map(\mathsf{A})(h_1)$ and the final state of h_1' is s.

$$
\begin{array}{l}
\forall\, h_1 : H_1 \bullet \\
\quad h_1.events \in \mathrm{seq}\, Event \Rightarrow \\
\quad\quad \forall\, s : s\_map(\mathsf{A})^{\sim}(\!|\ s\_map(\mathsf{A})(h_1.states(\#h_1.states))\ |\!) \bullet \\
\quad\quad\quad \exists\, h_1' : H_1 \bullet \\
\quad\quad\quad\quad h\_map(\mathsf{A})(h_1') = h\_map(\mathsf{A})(h_1) \wedge \\
\quad\quad\quad\quad h_1'.states(\#h_1'.states) = s
\end{array}
$$

Proof

The proof is by induction over the length of $h_1.events$.

(i) If $\#h_1.events = 0$ then $h_1.states(\#h_1.states) = h_1.states(1)$ must satisfy the predicate of the initial state schema of $C[A]$ using Method 1. Therefore, all states s such that $s\_map(\mathsf{A})(s) = s\_map(\mathsf{A})(h_1.states(1))$ must also satisfy the predicate of the initial state schema of $C[A]$ by Lemma C.3. Hence, there must exist a h_1' in H_1 such that $h\_map(\mathsf{A})(h_1') = h\_map(\mathsf{A})(h_1)$ and the final state of h_1' is s.

(ii) Assume the lemma is true for all h_1 in H_1 such that $\#h_1.events = n$ for some $n \geqslant 0$.

If $\#h_1.events = n+1$ then the state transition $(h_1.states(n+1), h_1.states(n+2))$ must be a transition of the event $h_1.events(n+1)$. Hence, for all states s' such that $s\_map(\mathsf{A})(s') = s\_map(\mathsf{A})(h_1.states(n+2))$, by Lemma C.6, there exists a state s such that $s\_map(\mathsf{A})(s) = s\_map(\mathsf{A})(h_1.states(n+1))$ and (s, s') is a transition of $h_1.events(n+1)$.

Since all pre-histories of h_1 are in H_1, there exists a history ph_1 in H_1 such that $ph_1 \in prehist(h_1)$ and $\#ph_1.events = n$. Therefore, there exists a history ph_1' in H_1 such that $h\_map(\mathsf{A})(ph_1') = h\_map(\mathsf{A})(ph_1)$ and $ph_1'.states(n+1) = s$ by the assumption. Therefore, there must exist a h_1' (which extends ph_1') in H_1 such that $h\_map(\mathsf{A})(h_1') = h\_map(\mathsf{A})(h_1)$ and the final state of h_1' is s'. Hence, the lemma is true for all h_1 in H_1 such that $\#h_1.events = n+1$. □

Lemma C.8

If the sequence of histories of a in a total history h_1 of $C[A]$ is in $h\_seq(\mathsf{A})$ then the sequence of behaviours of a in $h\_map(\mathsf{A})(h_1)$ is in $b\_seq(\mathsf{A})$.

Proof

Let hs denote the sequence of histories of a in h_1. If hs is in $h\_seq(\mathsf{A})$ then from the definition of $h\_seq(\mathsf{A})$,

$$hs \in \mathrm{dom}\ closure \wedge$$
$$closure(hs) \in \mathcal{TH}(\mathsf{A})$$

Let bs denote the sequence of behaviours of a in $h\_map(\mathsf{A})(h_1)$. Since $hs \in \mathrm{dom}\ closure$, from the definition of $closure$, the history of a in h_1 is either continually extended or, after some point, remains unchanged. Therefore, from the definition of $behav(\mathsf{A})$, the behaviour of a in $h\_map(\mathsf{A})(h_1)$ is either continually extended or, after some point, remains unchanged. Hence, from the definition of $b\_closure$, $bs \in \mathrm{dom}\ b\_closure$.

Furthermore, it follows that $b\_closure(bs) = behav(\mathsf{A})(closure(hs))$. Therefore, since $behav(\mathsf{A})(\!|\ \mathcal{TH}(\mathsf{A})\ |\!) = \mathcal{CR}(\mathsf{A})$, $b\_closure(bs) \in \mathcal{CR}(\mathsf{A})$.

Hence, from the definition of $b\_seq(\mathsf{A})$, $bs \in b\_seq(\mathsf{A})$. □

Lemma C.9

Let H_1 denote the set of safe histories of $C[A]$ derived using Method 1 and H_2 denote the set of safe histories of $C[A]$ derived using Method 2. If the sequence of behaviours of a in a history h_2 of H_2 is in $b\_seq(\mathsf{A})$ then there exists a history h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ such that the sequence of histories of a in h_1 is in $h\_seq(\mathsf{A})$ and h_1 is in H_1.

Proof

Let bs denote the sequence of behaviours of a in h_2. If bs is in $b\_seq(\mathsf{A})$ then from the definition of $b\_seq(\mathsf{A})$,

$$bs \in \operatorname{dom} b\_closure \wedge$$
$$b\_closure(bs) \in \mathcal{CR}(\mathsf{A})$$

Let h_a be a history in $\mathcal{TH}(\mathsf{A})$ such that $behav(\mathsf{A})(h_a) = b\_closure(bs)$. The existence of h_a is guaranteed since $behav(\mathsf{A})(\!|\ \mathcal{TH}(\mathsf{A})\ |\!) = \mathcal{CR}(\mathsf{A})$. Let h_1 be a history such that $h\_map(\mathsf{A})(h_1) = h_2$ and at each state in h_1 the history of a is a pre-history of h_a. The existence of h_1 is guaranteed since $\mathcal{TH}_{safe}(\mathsf{A})$ contains all pre-histories of h_a, and, hence, each history of a in h_1 is a possible safe history of A.

Let hs denote the sequence of histories of a in h_1. Since $bs \in \operatorname{dom} b\_closure$, from the definition of $b\_closure$, the behaviour of a in h_2 is either continually extended or, after some point, remains unchanged. Therefore, from the definition of $behav(\mathsf{A})$, the history of a in h_1 is either continually extended or, after some point, remains unchanged. Hence, from the definition of $closure$, $hs \in \operatorname{dom} closure$. Furthermore, $closure(hs) = h_a$ is in $\mathcal{TH}(\mathsf{A})$. Hence, from the definition of $h\_seq(\mathsf{A})$, $hs \in h\_seq(\mathsf{A})$.

Also, since any event of $C[A]$ which extends the behaviour of a (i.e. an event which applies an operation to a or changes a arbitrarily) can also extend the history of a by the same events, and any event of $C[A]$ which doesn't change the behaviour of a (i.e. an event which doesn't refer to a or changes a arbitrarily) can also leave the history of a unchanged, the sequence of events of $C[A]$ which result in h_2 using Method 2 can result in h_1 using Method 1. Hence, h_1 is in H_1. □

Lemma C.10

Let H_1 denote the set of safe histories of $C[A]$ derived using Method 1 and H_2 denote the set of safe histories of $C[A]$ derived using Method 2.

(a) If a history h_1 satisfies a history invariant in $C[A]$ using Method 1 then $h\_map(\mathsf{A})(h_1)$ satisfies the history invariant using Method 2.

(b) If a history h_2 is in H_2 and satisfies a history invariant in $C[A]$ using Method 2 then there exists a h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ such that h_1 is in H_1 and satisfies the history invariant using Method 1.

Proof

1) The history invariant does not refer to a.

Whether a history satisfies the history invariant is independent of the value of a.

(a) If h_1 satisfies the history invariant using Method 1 then $h\_map(\mathsf{A})(h_1)$ satisfies the history invariant using Method 2. Therefore, the history invariant satisfies part (a) of the lemma.

(b) If h_2 satisfies the history invariant using Method 2 then all h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ satisfy the history invariant using Method 1. Also, there exists a h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ which is also in H_1 by Theorem 5.1(b). Therefore, the history invariant satisfies part (b) of the lemma.

2) The history invariant is $\vec{a}$.

Using Method 1, h_1 satisfies the history invariant when the sequence of histories of a in h_1 is in $h\_seq(\mathsf{A})$. Using Method 2, h_2 satisfies the history invariant when the sequence of behaviours of a in h_2 is in $b\_seq(\mathsf{A})$.

(a) Since the sequence of histories of a in h_1 is in $h\_seq(\mathsf{A})$, the sequence of behaviours of a in $h\_map(\mathsf{A})(h_1)$ is in $b\_seq(\mathsf{A})$ by Lemma C.8. Therefore, the history invariant satisfies part (a) of the lemma.

(b) Since the sequence of behaviours of a occurring in h_2 is in $b\_seq(\mathsf{A})$, there exists a h_1 in $h\_map(\mathsf{A})^{\sim}(\!|\ \{h_2\}\ |\!)$ such that the sequence of histories of a in h_1 is in $h\_seq(\mathsf{A})$ and h_1 is in H_1 by Lemma C.9. Therefore, the history invariant satisfies part (b) of the lemma.

All other history invariants in $C[A]$ can be constructed from the history invariants above and, using the above results, can be shown to satisfy the lemma. □

Index of Definitions